2015-03

# Performance testing of GPU-based approximate matching algorithm on network traffic

Jimoh, Mujeeb B.

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/45198

# NAVAL POSTGRADUATE SCHOOL

**MONTEREY, CALIFORNIA**

# APPLIED CYBER OPERATIONS CAPSTONE PROJECT REPORT

**PERFORMANCE TESTING OF GPU-BASED APPROXIMATE MATCHING ALGORITHM ON NETWORK TRAFFIC**

by

Mujeeb B. Jimoh

March 2015

| | |
|---|---|
| Capstone Advisor: | Robert Beverly |
| Co-Advisor: | Michael McCarrin |

| | | | |
|---|---|---|---|
| **REPORT DOCUMENTATION PAGE** | | | *Form Approved OMB No. 0704–0188* |

| | | |
|---|---|---|
| **1. AGENCY USE ONLY *(Leave blank)*** | **2. REPORT DATE** March 2015 | **3. REPORT TYPE AND DATES COVERED** Capstone Project Report |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** PERFORMANCE TESTING OF GPU-BASED APPROXIMATE MATCHING ALGORITHM ON NETWORK TRAFFIC | **5. FUNDING NUMBERS** |
| **6. AUTHOR(S)** Mujeeb B. Jimoh | |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Naval Postgraduate School Monterey, CA 93943-5000 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)** N/A | **10. SPONSORING/MONITORING AGENCY REPORT NUMBER** **N/A** |

**11. SUPPLEMENTARY NOTES** The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ____N/A____.

| | |
|---|---|
| **12a. DISTRIBUTION / AVAILABILITY STATEMENT** Approved for public release; distribution is unlimited | **12b. DISTRIBUTION CODE** |

**13. ABSTRACT (maximum 200 words)**

Insider threat is one of the risks both government and private organizations have to deal with in protecting their important information. Data exfiltration and data leakage resulting from insiders' activities can be very difficult to identify and quantify. Unfortunately, existing solutions that efficiently check whether data moving across a network is known to be sensitive are not resilient to attackers that make changes—even trivial modifications—to the data prior to exfiltration.

This capstone examines the potential use of the *sdhash* approximate matching algorithm within the data exfiltration domain. *Sdhash* can be employed to look for active transfer of known sensitive files in network traffic, but in practice is hindered by the computational time required to check for known sensitive data. This research tested the performance of both the GPU and CPU implementation of *sdhash* to determine their suitability in high-network traffic environments such as the Department of Defense.

The results of this experiment showed that better performance is achieved with the GPU when comparing large data sets. For small data sets, the CPU and GPU implementations exhibited similar performance. Thus, *sdhash* in the GPU implementation would be suitable for the Defense Department's use.

| | | | |
|---|---|---|---|
| **14. SUBJECT TERMS** Data exfiltration, data leakage, insider threat, approximate matching, sdhash, GPU, CPU, similarity digest, and network capture. | | | **15. NUMBER OF PAGES** 71 |
| | | | **16. PRICE CODE** |
| **17. SECURITY CLASSIFICATION OF REPORT** Unclassified | **18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified | **19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified | **20. LIMITATION OF ABSTRACT** UU |

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

# PERFORMANCE TESTING OF GPU-BASED APPROXIMATE MATCHING ALGORITHM ON NETWORK TRAFFIC

Mujeeb B. Jimoh

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN APPLIED CYBER OPERATIONS**

from the

**NAVAL POSTGRADUATE SCHOOL**

**March 2015**

Author: Mujeeb B. Jimoh

Approved by:

| | |
|---|---|
| Robert Beverly<br>Capstone Project Advisor | Michael McCarrin<br>Project Advisor |

Cynthia Irvine
Chair, Cyber Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

# ABSTRACT

Insider threat is one of the risks both government and private organizations have to deal with in protecting their important information. Data exfiltration and data leakage resulting from insiders' activities can be very difficult to identify and quantify. Unfortunately, existing solutions that efficiently check whether data moving across a network is known to be sensitive are not resilient to attackers that make changes—even trivial modifications—to the data prior to exfiltration.

This capstone examines the potential use of the *sdhash* approximate matching algorithm within the data exfiltration domain. *Sdhash* can be employed to look for active transfer of known sensitive files in network traffic, but in practice is hindered by the computational time required to check for known sensitive data. This research tested the performance of both the GPU and CPU implementation of *sdhash* to determine their suitability in high-network traffic environments such as the Department of Defense.

The results of this experiment showed that better performance is achieved with the GPU when comparing large data sets. For small data sets, the CPU and GPU implementations exhibited similar performance. Thus, *sdhash* in the GPU implementation would be suitable for the Defense Department's use.

THIS PAGE INTENTIONALLY LEFT BLANK

# TABLE OF CONTENTS

# LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| BYOD | Bring Your Own Device |
| COTS | Commercial Off-the-Shelf |
| CPU | Central Processing Unit |
| CTPH | Context Triggered Piecewise Hash |
| CUDA | Compute Unified Device Architecture |
| DCM | Device Control Module |
| DISA | Defense Information System Agency |
| DLP | Data Loss Prevention |
| DOD | Department of Defense |
| EMAIL | Electronic Mail |
| EPO | EPolicy Orchestrator |
| ESSG | Enterprise Solution Steering Group |
| FIPS | Federal Information Processing Standard |
| GB | GigaBytes |
| GCC | GNU Compiler Collection |
| GDDR | Graphics Double Date Rate |
| GOTS | Government off-the-shelf |
| GPU | Graphical Processing Unit |
| GPGPU | General-Purpose Graphic Processing Unit |
| HBSS | Host-Based Security System |
| HIPS | Host Intrusion Prevention System |
| HPC | High Performance Cluster |
| IM | Instant Messenger |
| IP | Intellectual Property |
| IT | Information Technology |
| MA | McAfee Agent |
| MD5 | Message Digest 5 |
| MRSH-V2 | Multi-Resolution Similarity Hash Version 2 |
| NIDS | Network Intrusion Detection System |
| NIPRNET | Non-Classified Internet Protocol Routed Network |

| | |
|---|---|
| NIST | National Institute of Standard and Technology |
| NSA | National Security Agency |
| NSRL | National Software Reference Library |
| PA | Policy Auditor |
| PII | Personally Identifiable Information |
| RSD | Rogue System Detection |
| SDBF | Similarity Digest Bloom Filter |
| SDHASH | Similarity Digest Hash |
| SHA | Secure Hash Algorithm |
| SIPRNET | Secret Internet Protocol Routed Network |

# I. INTRODUCTION

The threat posed by insiders to governments, institutions and organizations continues to grow, yet not enough protections exist to combat this threat. A greater fraction of IT budgets goes toward providing perimeter defenses [1], while less is spent on new technologies that will provide better protection for an organization's important data. Despite rigorous background checks and lie-detector examinations (polygraphs) by governments and organizations, we are still seeing an increase in the number of IT administrators constantly abusing their privileged access by viewing or stealing sensitive data. Sensitive data include but are not limited to customer data, weapon design, intellectual property (IP), credit card information, and trade secrets. Data exfiltration and leakage can force a business to fold, and other consequences may include loss of money, loss of competitive advantage, loss of trust in government, and endangerment to national security.

The two types of insiders that pose a threat to an organization's sensitive data are inadvertent insiders and malicious insiders. An inadvertent insider is a trusted person with access to sensitive information who unintentionally discloses it. A malicious insider is defined as a current or former employee, contractor, or business partner who has or had authorized access to an organization's network, system, or data and intentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information and information systems [2]. There are a variety of reasons why a malicious insider will want to leak or steal sensitive information. These reasons include excessive debt, retaliation against the organization, family problems like divorce or marital conflict, inadequate safeguards and improper classification of sensitive data, inadequate organization policy, etc.

Even though many surveys suggest that the number of insider attacks is smaller than the number of outsider attacks [3], [4], the damage resulting from an insider's attack can be more dangerous, more devastating and more challenging to detect and prevent given the fact that insiders have legitimate access to an organization's network and information. A prominent example is that of National Security Agency (NSA) contractor employee Edward Snowden, who leaked details of a purported NSA program known as Prism—a classified program that allows the government to tap into the central servers of nine leading U.S. Internet companies and collect metadata (data about data) [5]. Another example of a recent insider attack is that of Morgan Stanley, the sixth-largest financial service company in the United States. One of its employees stole and posted account data for hundreds of thousands of its wealth management clients [6]. An additional example of insider threat is that of U.S. Army Private First Class Bradley Manning, an intelligence analyst with a security clearance, who allegedly downloaded classified files from military networks and leaked them to the anti-secrecy website WikiLeaks [7]. Despite the staggering number of data breach incidents by insiders being reported in media outlets, governments and organizations are still not doing enough to safeguard their important information. According to Park, enterprises and governments will fail to protect 75 percent of sensitive data by the year 2020 and by 2015, at least one more Snowden- or WikiLeaks-like event is likely to occur [8].

## A. MOTIVATION

Data exfiltration is the unauthorized transfer of sensitive information from a target's network to a location that a threat actor controls [9]. Advances in technology and always-on high-speed Internet connectivity have provided insiders many avenues by which they can easily exfiltrate or leak an organization's sensitive information. Insiders can transfer data over the network either by sending it as an attachment in electronic mail (email), Instant Messenger (IM), posting it on social media sites such as Facebook, Instagram, Twitter, LinkedIn, etc., or transferring it to cloud storage services such as Google Drive, Microsoft OneDrive, DropBox, Apple iCloud. In some organizations where employees are allowed to bring their own devices (BYOD), malicious insiders can also use this outlet to steal an organization's important data. Given the various avenues that can be employed to steal or disclose sensitive data, it becomes a daunting task for organizations to know when their important data are being exfiltrated.

## B. SCENARIO

Illustrated in Figure 1 is a real live scenario where a malicious insider used email to exfiltrate a highly classified sensitive document. This scenario occurred in a military unit, which had a keyword matching filtering system in place to detect exfiltration. As seen in the figure, the insider used the built-in Search and Replace function on Microsoft Word to replace some characters in the sensitive document (i.e., a for @, s for $, e for 3) and sent it as an attachment. The keyword matching filtering system failed to alert because it could not match exactly the defined keywords. As a result, the insider was able to successfully exfiltrate the sensitive document. Thus, a keyword matching filtering system as a way for organizations to protect their sensitive data is not good enough because of its reliance on exact keyword matching.

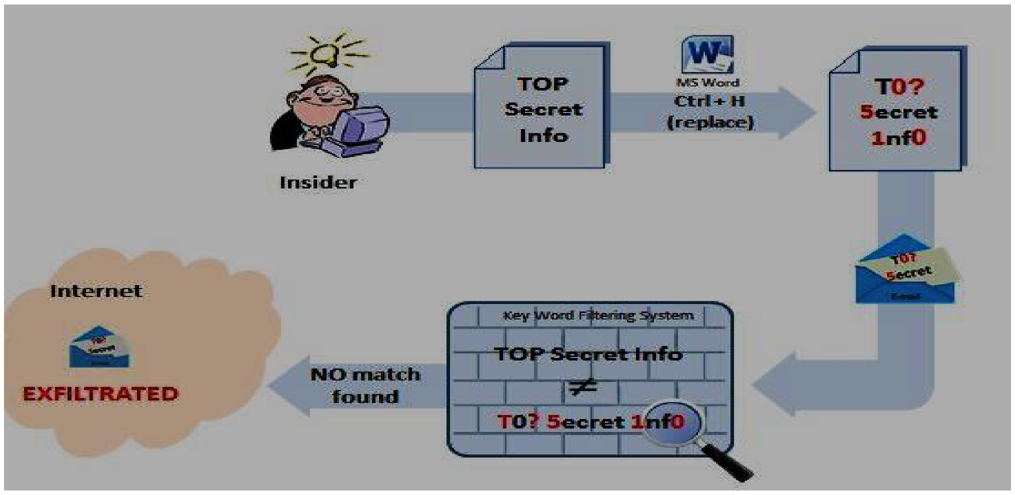


Figure 1. Diagram showing insider exfiltrating sensitive file

With this problem in mind, this study will employ an approximate matching algorithm known as *sdhash* in an effort to detect data exfiltration over the network, while measuring *sdhash* performance on both a Central Processing Unit (CPU)—which is responsible for normal processing of computer instructions, normally consisting of a few cores optimized for sequential serial processing—and a Graphic Processing Unit (GPU)—optimized for parallel processing, normally consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously, in order to determine which one will be suitable in a high network traffic environment such as the DOD.

The main questions this capstone seeks to answer are:

1. Which implementation of *sdhash's* approximate matching algorithm is practical in DOD networks (GPU vs CPU implementation)?

2. Can approximate matching completely stop data exfiltration/leakage?

## C. THESIS OUTLINE

The remainder of this paper is organized as follows: Chapter II describes existing solutions used by the DOD to prevent data exfiltration/leakage, describes previous work, provides background on approximate matching algorithms and examines three of the prominent ones. Chapter III explains the methodology used in carrying out the experiment. Chapter IV describes results of the experiment, while Chapter V concludes, presents research for future study, and provides limitations of approximate matching.

# II. BACKGROUND

## A. DEPARTMENT OF DEFENSE EXISTING SOLUTIONS TO DATA EXFILTRATION

### 1. HOST-BASED SECURITY SYSTEM

A host-based security system (HBSS) is a collection of flexible, commercial off-the-shelf (COTS) and government off-the-shelf (GOTS) applications designed to detect and counter, in real time, known cyber threats to the Department of Defense (DOD) [10]. It was conceptualized in 2005 by DOD Enterprise Solutions Steering Group (ESSG) and its initial rollout began in 2006. It is deployed on DOD enclaves such as Non-Classified Internet Protocol Routed Network (NIPRNET) and Secret Internet Protocol Routed Network (SIPRNET). Some of the HBSS application components include anti-virus, anti-spyware, Rogue System Detection (RSD), Host Intrusion Prevention (HIPS), Asset Baseline Monitor (ABM), Policy Auditor (PA), McAfee Agent (MA) and Device Control Module (DCM)—a subset of Data Loss Prevention (DLP). The DCM component of HBSS is what DOD uses as its solution to data exfiltration and leakage.

### 2. DATA LOSS PREVENTION (DLP)/DEVICE CONTROL MODULE (DCM)

Data Loss Prevention (DLP) is an approach used to detect, monitor, and protect confidential data at rest, in motion, and on the endpoint through deep content inspection and the constant monitoring of transactions occurring on each host or across the network [11]. Many security vendors such as McAfee, Symantec, TrendMicro, Sophos, etc., now offer DLP either as a standalone solution to data exfiltration/leakage or as part of a security suite like HBSS.

The Device Control Module (DCM), a component of the HBSS suite, is a subset of the McAfee product Data Loss Prevention. It is designed to prevent data exfiltration and leakage by preventing the unauthorized use of peripheral devices such as thumb drives and other removable storage. Its main function is to prevent insiders, whether malicious or inadvertent, from copying sensitive information into unauthorized removable drives.

The guide to creating HBSS DCM rules can be downloaded from Defense Information System Agency (DISA). The Information Assurance Manager (IAM) is responsible for following the guide to create block removable USB device rule and deploying the rule to all hosts (servers, workstations, and laptops) connected to the DOD network. After the rule has been applied to all hosts, any subsequent USB drives plugged into the host will be reported to a central management server known as the ePolicy Orchestrator (ePO) through McAfee Agent (MA).

The IAM is also responsible for the day-to-day upkeep of all HBSS components to ensure it is functioning properly. This responsibility includes downloading and applying patches from DISA repository, monitoring different events generated by different components of HBSS including DCM logged events, ensuring all hosts on the network have HBSS agents installed, etc. Figure 2 shows an ePO server web console dashboard where an administrator can login to see the status of different HBSS components including DCM activities. Information gathered here can be used for further investigation into the nature of security incidents.

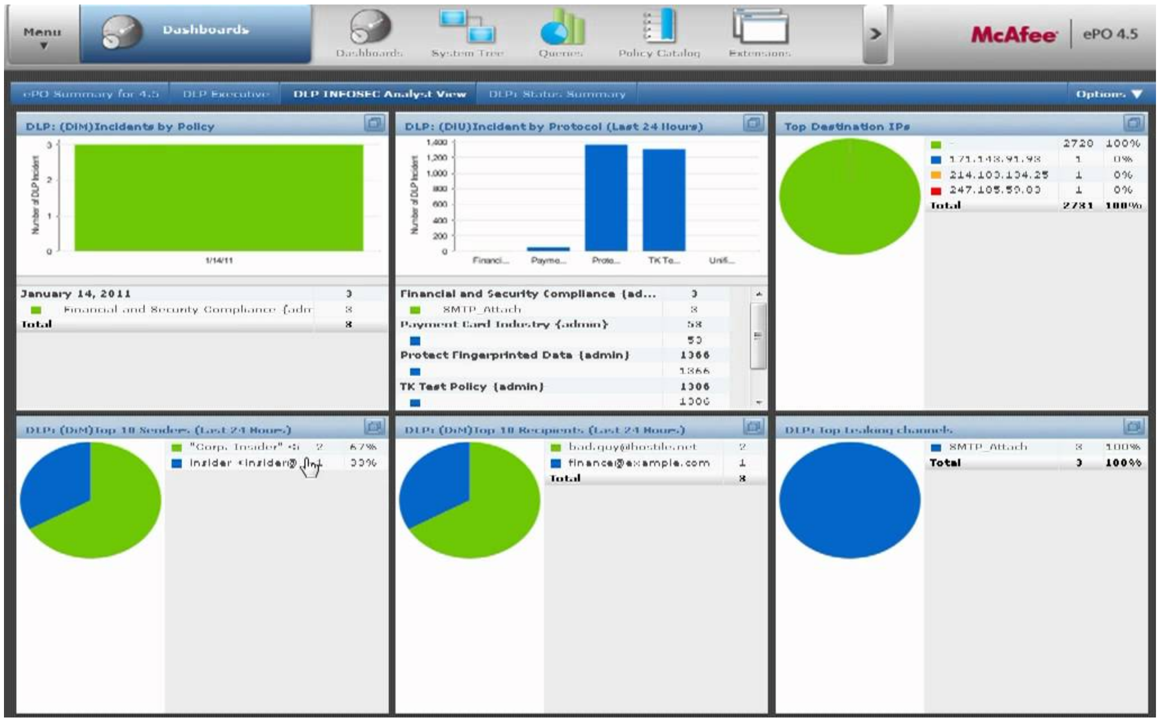


Figure 2. Example of McAfee ePolicy Orchestrator Management Console, from [12], showing Incidents by Policy, Protocols, and Destination IPs

Because DCM is implemented as a host-based data prevention solution, it presents some holes that could be exploited by malicious insiders. For one, this is only preventing exfiltration or leakage of data at endpoints or data at rest; it does not fully prevent exfiltration of data in motion, for instance sensitive files sent as an attachment in email or uploaded to cloud storage like Google Drive. Since DCM is installed as a service and relies on agent to send activities to centralized console, a rogue administrator can potentially stop the DCM service and disable the agent on a particular host in order to prevent it from reporting its activities. Even if the agent is reporting to the central server, the administrator still needs to be constantly monitoring the events in order to weed out false positives and false negatives—a practice that can be time and labour intensive. The centralized management of events also presents a single point of failure because if this server goes down some important events could be missed.

HBSS DCM is a step in the right direction for protecting the DOD's sensitive and classified information from being leaked or exfiltrated, but it has some weak points that can be exploited by a malicious insider who is aware of its presence. For these reasons, there is a need for either an all-encompassing data exfiltration detection and prevention solution or a solution that can work in conjunction with existing DCM (HIDS) in order to fully prevent or reduce exfiltration and leakage. Therefore, approximate matching based approaches promise an attractive method that can work in parallel with HBSS DCM in preventing exfiltration that occurs over the network.

## B. PREVIOUS WORK

History has shown that organizations and governments have been dealing with insider threat for quite a while and for this reason, different algorithms have been employed to protect sensitive data. Some of the prior efforts in data exfiltration prevention include pattern matching, keyword matching and cryptographic hashing.

### 1. Pattern Matching

Pattern matching is a technique used in automated data analysis to search for all occurrences of strings in a body of text. It is a problem of locating all occurrences of string ***x*** (the pattern) in another string ***t*** of length ***n*** (the text) [13]. Its objective is to identify the location of a specific text pattern within a larger body of text. Pattern matching is used in applications such as virus signature checking, Network Intrusion Detection Systems (NIDS), search-and-replace in word processors, web search engines, and spam filters. The two main types of pattern matching are exact pattern matching and regular expression pattern matching.

#### ***a. Exact Pattern Matching***

Exact pattern matching searches for occurrences of a single pattern in a body of text or binary data. Early NIDS, such as older release of snort [14], are based on exact pattern matching. Snort uses algorithms such as Rabin-Karp, Knuth-Morris-Pratt (KMP), and Boyer-Moore.

(1) Rabin-Karp

Michael O. Rabin and Richard M. Karp developed the Rabin-Karp algorithm in 1987 [15] with the intention to solve the shortcomings of the brute force algorithm. Their approach is to generate a digital signature (hash) of the pattern to find, then generate hash of all possible sub-strings of the text, and then compare both all at once [16]. Rabin-Karp's complexity is O(nm) where n is the length of the text and m is the length of the pattern, but in practice it's O(n+m). This algorithm saves time and is more efficient than brute force. Even though there are many faster algorithms, Rabin-Karp algorithm is very useful in detecting plagiarism because it is capable of performing multiple pattern matching.

(2) Knuth-Morris-Pratt (KMP)

Donald E. Knuth, James H. Morris, and Vaughan R. Pratt developed the Knuth-Morris-Pratt algorithm in 1977 in order to reduce the redundancy of the Rabin-Karp algorithm [17]. Their approach is to skip useless comparisons that happen in Rabin-Karp by first creating a partial match table of the pattern and then performing the search. It compares the characters in the pattern from left to right, using knowledge of previous characters compared. Knuth-Morris-Pratt's complexity is O(n+m) where n is the length of text while m represents length of pattern. This algorithm is faster than Rabin-Karp mainly because it does not need to keep going back to the beginning of the text whenever there is a mismatch in the comparison, which makes it suitable for processing large files. The following table shows the KMP algorithm:

```
KMP-Matcher(T,P)
    n = T.length
    m = P. length
    p = Compute-Prefix-Function(P)
    q = 0
    for i = 1 to n
         while q > 0 and P[q + 1] <> T[i]
               q = p[q]
               if P[q + 1] == T[i]
 q = q + 1
 if q == m
      print “Pattern occurs with shift” i - m
      q = p[q]
          return p
```

Table 1. Knuth-Morris-Pratt (KMP) algorithm

(3) Boyer-Moore

Robert S. Boyer and J Strother Moore developed Boyer-Moore algorithm in 1977 in order to improve the performance of pattern matching [18]. It is considered the most efficient pattern matching algorithm. Like KMP, this algorithm also pre-processes the pattern in order to compute a shift table. Unlike other pattern matching algorithms, it compares characters in the pattern from right to left and if a mismatch is found, it will compute how far the pattern should move to the right before another match is attempted. Boyer-Moore is used in text editors, such as in the Find and Replace function in Microsoft Word.

```
Boyer-Moore-Matcher(T,P,E)
  n = T.length
  m = P.length
  l = Compute-Last-Ocurrence-Function(P, m, E)
  y = Compute-Good-Suffix-Function(P, m)
  s = 0
      while s <= n – m
          do j = m
      while j > 0 and P[j] = T[s + j]
          do j = j – 1
if j = 0
print “Pattern occurs at shift” s
s = s + y[0]
else
s = s + max(y[j],j - l[T[s+j]]) □□ n mO□
```

Table 2. Boyer-Moore algorithm

### b. *Regular Expression Pattern Matching*

Regular expression (shortened to “regex”) pattern matching is used to detect patterns in data. It provides more efficiency and flexibility over exact pattern matching by allowing the use of logical operators like “or” and “and” to specify specific context to match [19]. Regular expression is the pattern matching of choice employed in open source NIDS such as Snort[1] and Bro.[2] In preventing data exfiltration, organizations employ this kind of technique by first defining patterns to look for in outgoing network traffic. For instance, the pattern for a social security number can be defined like this: xxx-xx-xxxx (x represents decimal number between 0 and 9). An NIDS based on regular expression pattern matching is employed at the network perimeter to look for those sensitive files that contain defined patterns and either block them from leaving the network or record them in the log [19].

[1] See http://manual.snort.org/

[2] See https://www.bro.org/documentation/index.html

### 2. Keyword Matching

Keyword matching involves developing some important words and putting them into a database that can be searched. These important words can be collected from sensitive files that need to be protected. The IDS can then be configured to look for files with those words in network traffic and either block any files with those words or log them. One of the problems with this is that some files that are not deemed sensitive may contain sensitive words and this can lead to high false positive rate. Also the amount of effort used to make up these words can be very high. Additionally, a malicious insider who is aware of the presence of a keyword matching system can defeat it by substituting characters in the file.

### 3. Cryptographic Hash

Cryptographic hash functions are used to prove that data has not been modified from its original version using the data's digital signature (digest). A cryptographic hash function will take data of arbitrary size as input and produce a fixed size digest as output. These digests can then be used to detect an unauthorized modification of data by comparing the digest of the original hash to the new hash. One of the important characteristics of cryptographic hash functions is that they are one-way functions, meaning the digest they generate is irreversible: one cannot determine the original data from the hash. There are many hash functions in use today, but the most widely used are Message Digest 5 (MD5) and Secure Hash Algorithm (SHA), which is a Federal Information Processing Standard (FIPS) approved cryptographic hash for Federal agencies [20].

A cryptographic hash is good for proving that two files are identical, but it is not suitable for detecting similarities between files. The problem with cryptographic hashing in terms of using it for data exfiltration prevention is its "avalanche effect." The avalanche effect describes a situation whereby a single bit flipped in an input to the hash function will result in a totally different output digest. As seen in Figure 3, two totally different digests were generated due to the fact that first input used a lower case "a" and the second input used an upper case "A." Because of this, an NIDS based on cryptographic hashes, employed by organizations to look for sensitive files transmitted across the network, can be deceived by a malicious insider who changes the characters in the sensitive files before exfiltration, knowing that the NIDS will not be able to match the hash of known data to the modified hash. Therefore, we need a solution that will be resilient to simple character modification. To improve the resilience of data exfiltration against modification attacks, this capstone considers using the *sdhash* approximate matching algorithm.

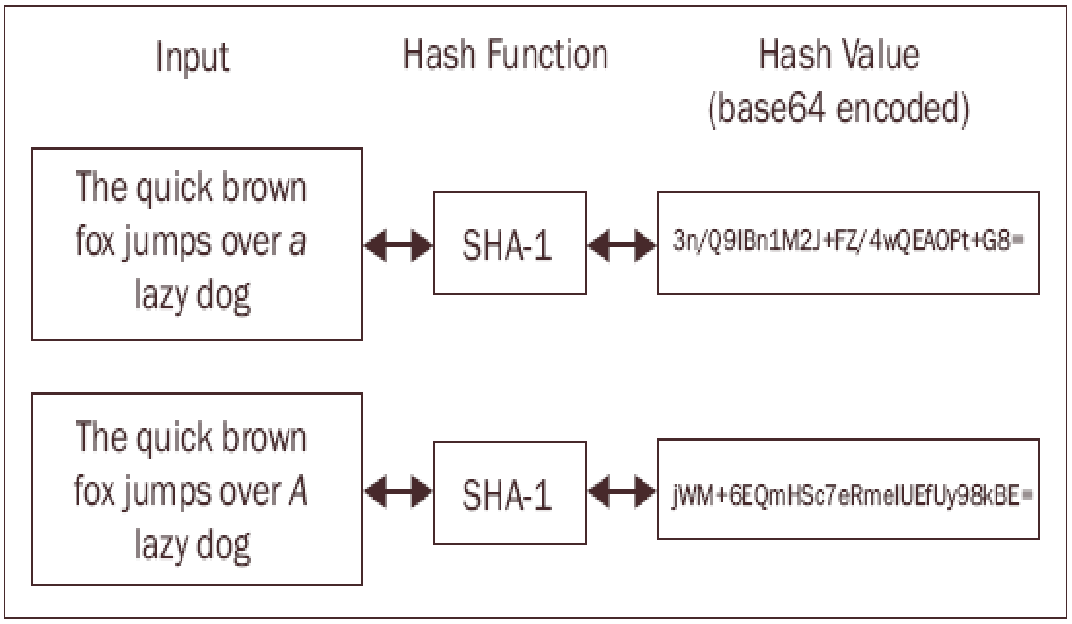

Input
Hash Function
Hash Value (base64 encoded)
The quick brown fox jumps over a lazy dog
SHA-1
3n/Q9IBn1M2J+FZ/4wQEAOPt+G8=
The quick brown fox jumps over A lazy dog
SHA-1
jWM+6EQmHSc7eRmeIUEfUy98kBE=

Figure 3. Example of SHA-1 Hash Function, from [21]

## C. APPROXIMATE MATCHING

Approximate matching is a generic term describing any technique designed to identify similarities between two digital artifacts such as files or images [22]. It can be employed to correlate complex and unstructured data that have certain amount of byte-level similarities. It does not rely on exact matching; it relies on finding similarities between two given files by comparing their 1s and 0s (byte level). Its applications include data filtering, security monitoring, digital forensics, malware detection, document versioning, etc.

Various research has been done on the uses of approximate matching, including Gupta's analysis of *sdhash* to detect files in network traffic [23]. Gupta's work examined both *sdhash* and mrsh-v2 approximate matching algorithms in detecting presence of known files (sensitive file) in network traffic. He argued that even though both can be used to detect the presence of known files in network traffic, *mrsh-v2* has better processing time.

Kornblum's context triggered piecewise hashing (CTPH) [24] involves breaking up a file into pieces (chunks), generating a 6-bit hash for each chunk using a rolling hash and then concatenating the hashes in order to produce the file digest. His idea is to combine a context triggered rolling hash and a traditional hashing algorithm to identify known files that have been modified or deleted.

Additional research that had been done on the usage of approximate matching is the work of Vassil Roussev in Data Fingerprinting with Similarity Digests [25]. Roussev's idea is to identify statistically-improbable features (features that are unique to data object such as file) and use them to generate similarity digest as opposed to randomized feature selection pioneered by Rabin in 1981. He argued that the use of similarity digest allows queries to be answered approximately, thereby providing a measure of correlation as opposed to cryptographic hashes that only support yes or no answers to digest queries. Another previous work on approximate matching is the work of Vassil Roussev and Candice Quates in Content Triage with Similarity Digests: The M57 case study [26]. Their work involved using *sdhash* to identify traces of data objects such as files, disk blocks, and network packets inside a bigger object of arbitrary size, such as disk image or network capture.

However, not much research has been done on the implementation of approximate matching on GPUs. Depending on the size of an organization, billions of packets can traverse a network in a given day. Most intrusion detection/prevention system employed on the network perimeter for filtering purposes have a hard time keeping up with the amount of network traffic because of the computationally intensive nature of comparing and matching signatures, strings, and hashes. Because of the special processing power of

GPUs (more memory bandwidth and more transistors for calculation), they can be employed on security devices to speed up hash lookup as well as string and signature comparison.

### 1. Byte-Wise Approximate Matching Algorithms

Three of the prominent algorithms in the field of byte-level approximate matching are *ssdeep*, *mrsh-v2*, and *sdhash*. Each has its advantages and disadvantages. The focus of this research is on *sdhash*, an approximate matching algorithm developed by Vassil Roussev and Candice Quates. A brief description of *ssdeep*, mrsh-v2, and *sdhash* is provided below.

#### *a. Ssdeep*

*ssdeep* was developed by Jesse Kornblum in 2006 [24]. It is used for producing context triggered piecewise hashes (CTPH), also called fuzzy hashes. His main contribution is to combine a context triggered rolling hash and a traditional hashing algorithm to identify known files that have been modified or deleted. His idea was, rather than generating a single hash for the entire file, a hash can be generated for many discrete fixed-size segments (chunks) of the file. For example, one hash is generated for the first 512 bytes of input, another hash for the next 512 bytes, and so on.

Hashes of each chunk are generated using a rolling hash. The resulting hashes are then concatenated to produce a similarity digest. In order to know where to start and stop traditional hashing of the chunks, this algorithm uses the rolling hash to identify a trigger point. A trigger point, which occurs at the end of a chunk is identified using a window size of 7 bytes that moves through the whole input. After a trigger point is identified, a traditional hash is carried out. The traditional hashing (non-cryptographic hash) was based on Fowler/Noll/Vo (FNV) hash.

#### *b. Mrsh-v2*

Multi-resolution similarity hashing version 2 (mrsh-v2) is an updated version of MRSH. It was proposed by Frank Breitinger and Harald Baier [27]. It borrowed design elements from both *ssdeep* and *sdhash*. Its approach is to break up an input, i.e., file, into fragments (chunks) and hash each chunk using a rolling hash. The resulting hashes are put into Bloom filters in order to save space and increase comparison efficiency. *Mrsh-v2* operates in two modes: regular mode and f-mode. Regular mode is used to identify similar files, while f-mode is used for detecting file fragmentation [28].

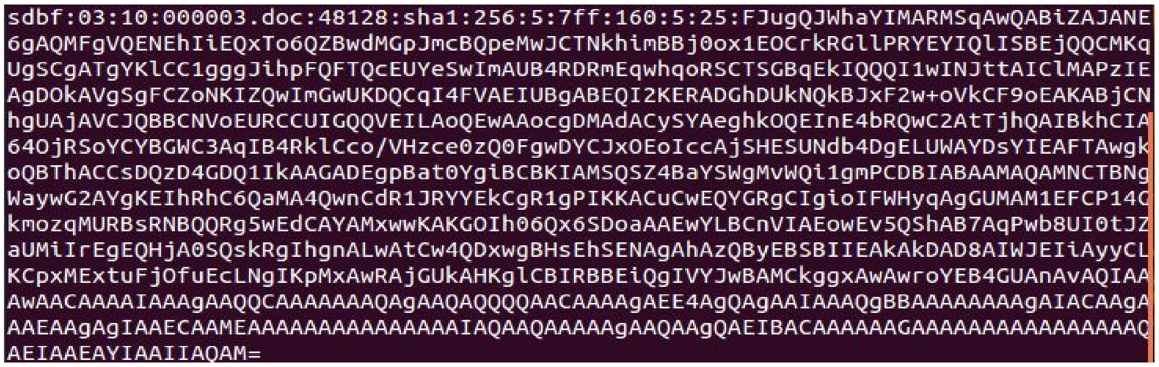
### *c. Sdhash*

*Sdhash* stands for similarity digest hash and it was developed by Vassil Roussev in 2010 [26]. It is an algorithm that allows two arbitrary blobs of data to be compared for similarity based on common strings of binary data [29]. *Sdhash's* approach is to identify statistically-improbable features—features that are least likely to occur in other data objects by chance and use them to generate similarity digests. Each of the features is hashed using the cryptographic hash function SHA-1 and the resulting hashes are put into a series of Bloom filters, which are a space-efficient set representation. In order to carry out comparison between two digital artifacts, their digests can be compared. *Sdhash* applications include identification of embedded objects, identification of code versions, identification of related documents, and correlation of network fragments.

*Sdhash* works in two modes namely continuous mode and block mode. Continuous mode is used to generate signatures for inputs less than 16MiB, which means the algorithm will continue adding unique features from the input to a Bloom filter until it reaches a saturation point set at 160 elements, at which point a new Bloom filter is created. Block mode is used for inputs greater than 16MiB. In this mode, the input is split into fixed-size blocks, which by default is set to 16KiB, and each block gets assigned to a separate Bloom filter. This default block value can be changed using the --block-size (-b) option in the algorithm. Features are selected from each block and added to that block's Bloom filter until either all the features are added or the filter reaches a saturation point of 192 elements [30]

(1) Digest generation

*Sdhash* digests can be generated using a single file or multiple files. The digest of a single file can be generated by typing *sdhash* then the file name. For example, ***sdhash myfile.doc***. The --*target-list (-f)* option, which will generate sdbf file (digests) from a list of filenames, is convenient for generating digests for multiple files. Sdbf stands for similarity digest bloom filter. It is a bit vector used for space efficient set representation. --*deep (-r)* option can also be used if the files are contained in directories. The minimum file size that can be use as input to *sdhash* is 512 bytes. Any input files less than 512 bytes will be silently skipped unless --*verbose (-v)* is specified as an option. Unless the --*output (-o)* option is used to direct digest output to the files specified, *sdhash* will print the digest to the standard output (screen). Each line of the digest consists of several header fields separated by semicolons, followed by base64-encoding of the digest data [29]. See sample output in Figure 4.



```
sdbf:03:10:000003.doc:48128:sha1:256:5:7ff:160:5:25:FJugQJWhaYIMARMSqAwQABiZAJANE
6gAQMFgVQENEhIiEQxTo6QZBwdMGpJmcBQpeMwJCTNkhimBBj0ox1EOCrkRGllPRYEYIQlISBEjQQCMKq
UgSCgATgYKlCC1gggJihpFQFTQcEUYeSwImAUB4RDRmEqwhqoRSCTSGBqEkIQQQI1wINJttAIClMAPzIE
AgDOkAVgSgFCZoNKIZQwImGwUKDQCqI4FVAEIUBgABEQI2KERADGhDUkNQkBJxF2w+oVkCF9oEAKABjCN
hgUAjAVCJQBBCNVoEURCCUIGQQVEILAoQEwAAocgDMAdACySYAeghkOQEInE4bRQwC2AtTjhQAIBkhCIA
64OjRSoYCYBGWC3AqIB4RklCco/VHzce0zQ0FgwDYCJxOEoIccAjSHESUNdb4DgELUWAYDsYIEAFTAwgk
oQBThACCsDQzD4GDQ1IkAAGADEgpBat0YgiBCBKIAMSQSZ4BaYSWgMvWQi1gmPCDBIABAAMAQAMNCTBNg
WaywG2AYgKEIhRhC6QaMA4QwnCdR1JRYYEkCgR1gPIKKACuCwEQYGRgCIgioIFWHyqAgGUMAM1EFCP14C
kmozqMURBsRNBQQRg5wEdCAYAMxwwKAKGOIh06Qx6SDoaAAEwYLBCnVIAEowEv5QShAB7AqPwb8UI0tJZ
aUMiIrEgEQHjA0SQskRgIhgnALwAtCw4QDxwgBHsEhSENAgAhAzQByEBSBIIEAkAkDAD8AIWJEIiAyyCL
KCpxMExtuFjOfuEcLNgIKpMxAwRAjGUkAHKglCBIRBBEiQgIVYJwBAMCkggxAwAwroYEB4GUAnAvAQIAA
AwAACAAAAIAAAgAAQQCAAAAAAAQAgAAQAQQQQAACAAAAgAEE4AgQAgAAIAAAQgBBAAAAAAAAAgAIACAAgA
AAEAAgAgIAAECAAMEAAAAAAAAAAAAAAAIAQAAQAAAAAgAAQAAgQAEIBACAAAAAAGAAAAAAAAAAAAAAAAQ
AEIAAEAYIAAIIAQAM=
```

Figure 4. Sample sdhash digest standard output.

From the output in Figure 4, each colon-delimited header is broken down in Table 3.

| Header | Meaning |
|---|---|
| sdbf | Sdbf’s magic string |
| 03 | Sdbf version number |
| 10 | Number of characters in input name |
| 000003.doc | The input name |
| 48128 | Input size in bytes |
| Sha1 | Hash algorithm used |
| 256 | Size of Bloom filter in bytes |
| 5 | Number of independent hash functions |
| 7ff | Mask value for determining which bits of the 5 sub-hashes generated by splitting the SHA1 hash should be used to map a feature to the 256-byte filter |
| 160 | Maximum number of feature element allowed in a Bloom filter (160 for continuous mode and 192 for block mode) |
| 5 | Number of Bloom filters |
| 25 | Number of features in the last Bloom filter |

Table 3. *Sdhash* digest header breakdown.

THIS PAGE INTENTIONALLY LEFT BLANK

# III. METHODOLOGY

In Chapter II, I described some previous work that had been performed in the domain of approximate matching and explained three of the prominent approximate matching algorithms. In this chapter, I detail the steps involved in performing my experiment. The goal is to compare reference sets, which represent sensitive files to be protected and target sets, which represent files captured over the network. These comparisons will be performed using CPU and GPU implementations of *sdhash* while measuring the time. To do this, I used the following six steps:

1. Request CPU and GPU nodes from an HPC cluster
2. Download and compile *sdhash* code on the given node
3. Download data files (GovDocs)
4. Generate digests for both reference and target sets using *sdhash*
5. Use the CPU implementation of *sdhash* to compare the reference set to the target set.
6. Use the GPU implementation of *sdhash* to compare the reference set to the target set.

## A. REQUESTING A CPU AND A GPU NODE FROM THE HPC CLUSTER

To carry out this experiment, the first thing I did was request a CPU node and a GPU node from the NPS "Hamming" High Performance Computing (HPC) cluster. Hamming is a supercomputer at the Naval Postgraduate School (NPS) designed for computationally intensive research for both students and faculty. It is named after a renowned mathematician Richard Hamming, who was a Professor of Computer Science at NPS from 1976 to 1998. Hamming contains over 2,000 computing cores. It has been used in student theses involving weather forecasting, polar ice prediction, modelling of helicopter rotors, data mining (extracting important data from very large datasets), and solving complicated mathematical equations.

To request a node, a *qsub*[3] job has to be submitted to the cluster. In the qsub job, one has to specify resources needed to complete the job. These resources include the

[3] See http://docs.adaptivecomputing.com/torque/4-0-2/Content/topics/commands/qsub.htm.

number of processors, amount of memory, amount of time, etc. For example, I submitted two *qsub* jobs, one requesting the CPU node with 64 cores and 2GB of memory per core (128GB) and the other requesting the GPU node with 8 GPUs. An example of each of the qsub jobs is given below.

```
qsub -I –l nodes=1:ppn=64,pmem=2gb,walltime=24:00:00,naccesspolicy=singleuser,hostlist=compute-7-27
```

```
qsub -I –l nodes=1:gpus=8,walltime=24:00:00,naccesspolicy=singleuser,hostlist=compute-8-17
```

Each option in the qsub job is explained below

- *I option* – requests the job to be run interactively.
- *l option* – specifies the resources required for the job
- *ppn* option – specifies number of processors per node. For the above job, a CPU node with 64 cores was requested.
- *walltime* option – specifies how long the job is going to be running on the given node. It specifies in *hh:mm:ss* format. The above jobs requested 24 hours
- *naccesspolicy* option – reserves the node for the requester alone. It allows no other user's job to be running on the given node. This is useful for measuring timing information.
- *gpus* option – is used to request GPUs. A node with 8 GPUs was requested for the job shown above.
- *pmem* option – specifies the memory per processor in GB. A CPU node with 128GB of memory was requested for this experiment.
- *hostlist* option – specifies a particular node out of the HPC node inventory.

The above jobs told the HPC cluster which specific resources I needed to do the comparisons. The HPC looked up the nodes specified in the jobs to see if they were available. Theoretically, if they were, it would come back and give me the node. If they were not, it would put the jobs in the queue until the resources were available to fulfil the jobs. In this case, the resources were available and the HPC gave me the nodes that I requested. The details of the nodes are explained below.

*CPU Node*

This Hamming node, Compute-7-27, is running CentOS release 6.6 with Linux version 2.6.32-431.20.3.el6.x86_64 and GNU Compiler Collection (GCC) 4.4.7. It includes 64 cores AMD Opteron™ Processor 6274[4] running at 1400 MHz (1.4GHz), with 128GB of memory (See Appendix B).

*GPU Node*

This Hamming node, Compute-8-27, is running CentOS release 6.6 with Linux version 2.6.32-431.20.3.el6.x86_64 and GCC compiler 4.4.7. It includes 8 GeForce GTX Black TITAN NVIDIA GPUs with 2880 CUDA[5] cores running at 889MHz. It also includes an on-board 6GB of GDDR5 memory [31] (See Appendix C).

Before I go further, it is important to point out some of the differences between a CPU and a GPU.

*Central Processing Unit (CPU)*

A CPU is a general-purpose processor that is considered the brain of a computer. It has four primary functions: fetch, decode, execute, and writeback. CPUs are developed and optimized for sequential serial processing. They contain more control hardware for tasks such as allocating memory and therefore reduce space available on the chip for calculations. CPUs normally comprise only a few cores with lots of cache memory.

*Graphic Processing Unit (GPU)*

GPUs were originally designed for rendering graphics, but have evolved to the point where many other real-world applications are being implemented on them. The development of Compute Unified Device Architecture (CUDA) parallel programming

[4] See http://www.cpu-world.com/CPUs/Bulldozer/AMD-Opteron%206274%20OS6274WKTGGGU.html.

[5] See http://www.nvidia.com/object/cuda_home_new.html.

language by NVIDIA has made it easy for programmers to write code that will harness the power of GPUs. Also the introduction of General-purpose Computing on Graphics Processing Units (GPGPU) has allowed computationally intensive research to be carried out using the GPU's parallel processing power. Some of the benefits of GPUs over CPUs include more computing power, larger memory bandwidth, and less power consumption.

## B. DOWNLOADING AND COMPILING *SDHASH* CODE

This experiment used *sdhash-3.4*. Even though, as at the time of this writing, *sdhash-4.0* was released, it was designated as an experimental version; and therefore it was considered not stable enough to be used for this experiment. *Sdhash-3.4* was downloaded from GitHub.[6] Because of administrative restrictions on the HPC supercomputer, I could only compile the code in my home directory. For the code to work on the GPU node, the CUDA 6.5 toolkit, NVIDIA drivers, and GCC compiler 4.4.7 were installed. For the CPU node, the GCC compiler was installed. After all the necessary libraries and dependencies were installed, I used the included Makefile in *sdhash* to compile the code.

## C. DOWNLOADING DATA FILES

For this experiment, I used the GovDocs[7] corpus, which is a collection of files with various file formats collected by crawling through U.S. Government websites and made freely available for research. The corpus was downloaded from http://digitalcorpora.org/ corpora/govdocs. It includes Word documents, Adobe pdfs, jpegs, html files, text files, PowerPoint files, gifs, Excel files, and so on. It is made up of 1000 directories with approximately 986 files in each directory, making a total of 986,278 files, totalling 468GB in size. The directories are numbered from 000 to 999. Files in each directory were named with a 3-digit directory number, a 3-digit file number, and the extension. For example, the 30th file in the 2nd directory, which happened to be a

---

[6] Available at https://github.com/sdhash/sdhash.

[7] Available at http://digitalcorpora.org/corpora/govdocs.

pdf file, was named as 001029.pdf. Table 4 shows the distribution of file types in GovDocs corpus.

| Format | doc | gif | html | jpg | pdf | Txt | xls | ppt |
|---|---|---|---|---|---|---|---|---|
| Count | 76,779 | 36,302 | 214,566 | 109,233 | 231,232 | 78,285 | 62,672 | 49,917 |

| Format | csv | ps | log | rtf | gz | xml | Pps | dbase3 | png | unk |
|---|---|---|---|---|---|---|---|---|---|---|
| Count | 18,360 | 28,826 | 9,976 | 1,125 | 13,725 | 33,458 | 1,619 | 2,601 | 4,125 | 5,186 |

Table 4. Distribution of file types in the GovDocs corpus

## D. GENERATING DIGESTS

The files in the GovDocs corpus were used to simulate both sensitive files to be protected and files captured over the network. For the purpose of comparison, I created two sets of similarity digests. The first is the reference set, which contains *sdhash* digests of simulated sensitive files. The second is the target set, which contains *sdhash* digests of simulated files captured over the network.

### 1. Reference Set

Because I want to do the comparison of reference sets and target sets in incremental order, I created five reference sets of varying sizes. To do this, I created five directories and named them as reference_250, reference_500, reference_750, reference_1000, and reference_2000. The number prepended to each named reference represents the number of files contained in that reference set. For example, reference_250 contained 250 files; reference_500 contained 500 files, and so on. In order to add files to each of my reference set directories, I first combined all the files in each of the GovDocs corpus directories into a single directory called All_GovDocs. I accomplished this by first using the ***find*** command to locate any files in the GovDocs directories and then the ***cp*** command to copy them to All_GovDocs directory. Here is the exact command:

```
find . –name *.* -exec cp –t ../All_GovDocs {} +
```

The command break down is:

- find – searches directories recursively downward from my current location denoted by the dot (.).
- -name – specifies the name of the file to find. In this command I used any files with any extension as denoted by *.*
- -exec cp – this denotes the action to perform on the files being found by *find* command. In this case I want to copy them.
- -t – this specifies the directory to copy the files to. I used ../ to tell the *cp* command to go up one directory from my current directory and put the copied files in All_GovDocs directory.
- {} – is a place holder that will be replaced by the files found.
- + – this prevents overflow of argument that the *cp* command can handle

After I combined all the files in the GovDocs directories into one directory, I copied the appropriate number of files to each of my reference sets. For example, I copied 250 files into my first reference set called reference_250 by doing the following:

*cp {000000..000249}.* ../reference_250*

Note: This command was executed from the All_GovDocs directory

The *cp command* copied files starting from 000000 to 000249 from the All_GovDocs directory to reference_250 directory. ***The wildcard .**** indicates that it doesn't matter what the extension of the files are. **../** will go up one directory from my current working directory (All_GovDocs) and find reference_250 directory where the copied files will be placed. I repeated this process to copy files to the rest of my reference sets with the following commands:

*cp {000000..000499}.* ../reference_500*
*cp {000000..000749}.* ../reference_750*
*cp {000000..000999}.* ../reference_1000*
*cp {000000..001999}.* ../reference_2000*

After I added files to my five reference sets, I generated similarity digests for each by running *sdhash* against each reference directory in order to produce a Similarity Digest Bloom Filter (sdbf) file. For example, to generate the sdbf file for reference_250, I ran the following *sdhash* command on the CPU node:

*sdhash -r reference_250 > reference_250.sdbf*

- *-r instructs sdhash to generate an sdbf from the directory given (reference_250)*
- *> redirects the output to a file called reference_250.sdbf*

Note: I did not measure time performance for generating similarity digests between the CPU and GPU because the GPU implementation of *sdhash* does not include the option to generate digests; it can only do comparison.

Table 5 shows the name of my reference sets, the number of digests contained in each, the total size of the files in each reference before I ran them through *sdhash* and the total size of the digests after I ran them through *sdhash*.

| Name of Reference Set | Number of Files/Digests in Each Reference Set | Size before ***sdhash*** (Raw Files) (MB) | Size after ***sdhash*** (Digests) (MB) |
|---|---|---|---|
| Reference_250.sdbf | 250 | 222 | 4 |
| Reference_500.sdbf | 500 | 322 | 8.4 |
| Reference_750.sdbf | 750 | 477 | 13 |
| Reference_1000.sdbf | 1000 | 733 | 21 |
| Reference_2000.sdbf | 2000 | 1343 | 39 |

Table 5. Reference sets with number of digests per reference set, total file size before *sdhash* and total digest size after *sdhash*

### 2. Target Set

Different networks will have different properties, so I cannot come up with a number that works for all of them. For this reason, I arbitrarily came up with the number of files to include in my target sets. I first chose 30,000 files from the GovDocs corpus to include in my first target set. The way I did this was similar to the way I created my reference sets. From creating my reference sets, I already had all files in each directory of the GovDocs corpus combined into one directory called All_GovDocs. Therefore, I copied the first 30,000 files from the All_GovDocs directory to a directory called target_30. To accomplish this, I used the following command from inside the All_GovDocs directory:

*cp {000000..029999}.* ../target_30*

The *cp command* copied files starting from 000000 to 029999 from the All_GovDocs directory to target_30 directory. ***The wildcard .**** was used to ignore the extention of the files. **../** indicated to go up one directory from my current working directory (All_GovDocs) and find target_30 directory where the copied files will be placed. I repeated this process to copy files to the rest of my target sets with the following commands:

*cp {000000..029999}.* ../target_30*

*cp {000000..035999}.* ../target_36*

*cp {000000..182999}.* ../target_183*

*cp {000000..199999}.* ../target_200*

It is important to know that all these target sets are essentially subset of one another because the copied files still remained in the All_GovDocs directory. This means the files contained in the target_30 are also part of target_36 and so on.

After I added files to my five target sets, I generated similarity digests for each by running *sdhash* against each target directory in order to produce a Similarity Digest Bloom Filter (sdbf) file. For example, to generate the sdbf file for my first target set called target_30, I ran the following *sdhash* command on the CPU node:

*sdhash -r target_30 > target_30.sdbf*

Table 6 shows the name of target sets, the number of digests in each, the size of the directory before *sdhash* and size of the digest after *sdhash*.

| Name of Target Set | Number of Files/Digests In Each Target Set | Size before *sdhash* (Raw Files) (GB) | Size after *sdhash* (Digests) (MB) |
|---|---|---|---|
| Target_30.sdbf | 30,000 | 16 | 496 |
| Traget_36.sdbf | 36,000 | 19 | 607 |
| Traget_65.sdbf | 65,000 | 34 | 1100 |
| Target_183.sdbf | 183,890 | 100 | 2400 |
| Target_392.sdbf | 392,078 | 200 | 5100 |

Table 6. Table showing name of target sets, number of digests per target set, total file size before *sdhash* and total size of digests after *sdhash*

## E. COMPARING REFERENCE SET AND TARGET SET USING THE CPU IMPLEMENTATION OF *SDHASH*

After I finished generating *sdhash* digests for both the reference sets and the target sets, I compared them using the CPU implementation of *sdhash* algorithm. The comparisons were conducted between each of the five reference sets and each of the five target sets using CPU. To do this, I compared each of my five reference sets and the first target set (target_30.sdbf) five times in order to measure the time accurately (what I mean by measuring the time accurately is that I don’t want too much disparity between each time measurement). For example, I did comparisons between reference_250.sdbf and target_30.sdbf, between reference_500.sdbf and target_30.sdbf, between reference_750.sdbf and target_30.sdbf, between reference_1000.sdbf and target_30.sdbf, and between reference_2000.sdbf and target_30.sdbf. Each comparison was performed five times and the time measurement was recorded for each. An example of the command used to do the comparison between the reference sets and the target sets using the CPU implementation of *sdhash* is as follows:.

*date >> time.txt; sdhash -c -p 64 reference_250.sdbf target_30.sdbf > 250_30_cpu.txt; date >> time.txt*

The command break down is:

- *date command - grabs the date and time and appends it to time.txt file.(**start time**)*
- *-c - runs sdhash in comparison mode.*
- *-p 64 - informs sdhash to use all available 64 cores in the node. (if this is not specified, sdhash automatically uses all available cores anyway)*
- *reference_250.sdbf - is the reference set that is being queried in target set.*
- *target_30.sdbf - is the target set being queried.*
- *250_30_cpu.txt - redirects the output to a text file called 250_30_cpu.txt.*
- *date command - outputs the date and time to time.txt file. (**end time**)*

The time it took to complete the CPU comparison was calculated by subtracting the start time from the end time. For instance, the CPU comparison of reference_250.sdbf and target_30.sdbf had a start time of 22:06:47 (hour:minute:seconds) and an end time of 22:08:44 (hour:minute:seconds). So to calculate the time it took to complete this comparison, I subtracted 22:06:47 from 22:08:44, which equals 117 seconds (1 minute and 57 seconds).

The computing resources used for all CPU comparisons was explained in A(1) of this chapter and the detailed results of each comparison are explained in the Results section of this paper.

## F. COMPARING REFERENCE SET AND TARGET SET USING THE GPU IMPLEMENTATION OF *SDHASH*

After comparing each of the five reference sets and each of the five target sets using the CPU implementation of *sdhash*, I carried out the same comparison using the GPU implementation of *sdhash*. This is essentially doing the same thing but instead of using the CPU to compare, I used the GPU. Using the GPU node, I conducted comparisons between each of the five reference sets and each of the five target sets. For instance, I compared reference_250.sdbf and target_30.sdbf, reference_500.sdbf and target_30.sdbf, reference_750.sdbf and target_30.sdbf, reference_1000.sdbf and target_30.sdbf and reference_2000.sdbf and target_30.sdbf. Each comparison was run five times and the time was recorded for each set of comparisons. I repeated the same process for target_36.sdbf, target_65.sdbf, target_183.sdbf, and target_392.sdbf. An example of command to run *sdhash* comparison in GPU is as follows:

```
date >> time.txt; sdhash-gpu  -r  reference_250.sdbf  -t  target_30.sdbf >  250_30_gpu.txt; date >> time.txt
```

The command break down is:

- *date command – grabs the date and time and append it to time.txt file. (**start time**)*
- *-r – specifies the reference set (reference_250.sdbf)*
- *-t – specifies the target set (target_30.sdbf)*
- *250_30_gpu.txt – redirects the output of the comparison to a text file called 250_30_gpu.txt.*
- *date command – outputs the date and time to time.txt file (**end time**)*

To calculate comparison time in GPU, I subtracted the start time from the end time. For example, one of the five comparisons between reference_250.sdbf and target_30.sdbf had a start time of 08:37:53 (hour:minute:seconds) and an end time of 08:39:26 (hour:minute:seconds). So to calculate comparison time, I subtracted 08:37:53 from 08:39:26, which equals 93 seconds (1 minute and 33 seconds).

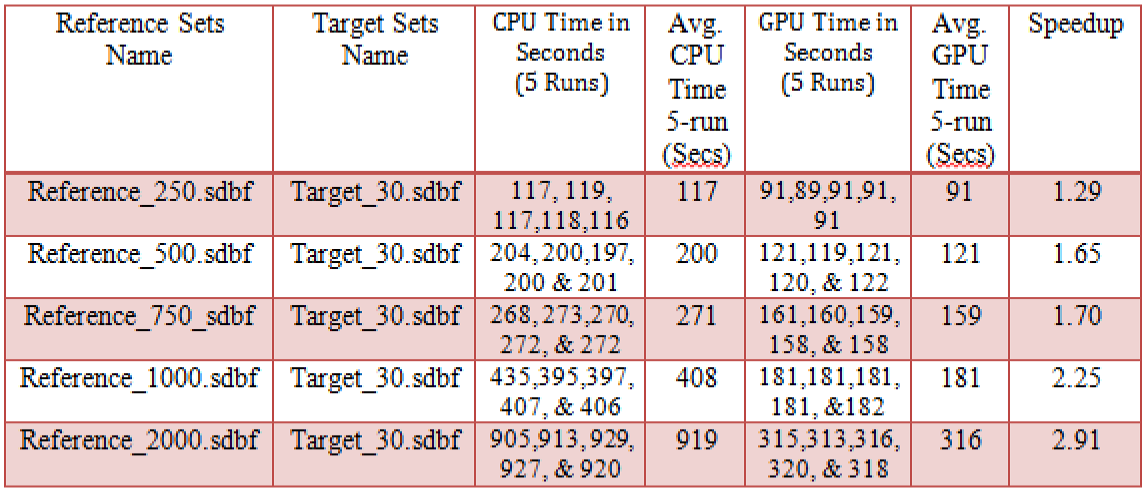
The computing resources used for all GPU comparisons were explained in A(2) of this chapter. The results of the comparisons are explained in the Results section of this paper.

# IV. RESULTS

Having generated digests of the reference sets and the target sets and compared them using the CPU and GPU implementation of *sdhash* in Chapter III, I will use this chapter to explain my results from the comparisons. For brevity, I put the results of each comparison in tables and charts. As I mentioned in the Methodology section, each of the five reference sets is compared to each of the five target sets five times, while taking time measurements of each run. To this end, I prepared five tables and five charts that show the results.

Included in each table are seven columns. The first column is the name of the reference sets. The second column is the name of the target sets. The third column is used to record CPU times from each of the five runs, measured in seconds. The fourth column is the average CPU time derived by adding each CPU time from each run and dividing the total by 5. For example, the average CPU time of comparing reference_250.sdbf and target_30.sdbf was calculated by adding 117, 119, 117, 118, and 116 and dividing the total by 5, which equals 117.4, and rounding down to 117 seconds. The fifth column is used to record GPU times from each of the five runs measured in seconds. The sixth column is the average GPU time derived by adding each GPU time from each run and dividing the total by 5. For example, the average GPU time of comparing reference_250.sdbf and target_30.sdbf was calculated by adding 91, 89, 91, 91 and 91 and dividing the total by 5, which equals 90.6 seconds, and rounding up to 91 seconds.

The seventh column is the speedup between CPU time and GPU time. To calculate this speedup, I divided the average CPU time by the average GPU time. For example, to calculate the speedup between the CPU comparison time and the GPU comparison time of reference_250.sdbf and target_30.sdbf, I divided the average CPU time 117 by the average GPU time 91, which equals 1.29. What this means is that GPU is 1.29x times faster than CPU when comparing reference_250.sdbf and target_30.sdbf. Table 7 shows the results of the first comparison between varying sizes of reference sets and target_30.sdbf.



| Reference Sets Name | Target Sets Name | CPU Time in Seconds (5 Runs) | Avg. CPU Time 5-run (Secs) | GPU Time in Seconds (5 Runs) | Avg. GPU Time 5-run (Secs) | Speedup |
|---|---|---|---|---|---|---|
| Reference_250.sdbf | Target_30.sdbf | 117, 119, 117,118,116 | 117 | 91,89,91,91, 91 | 91 | 1.29 |
| Reference_500.sdbf | Target_30.sdbf | 204, 200,197, 200 & 201 | 200 | 121,119,121, 120, & 122 | 121 | 1.65 |
| Reference_750_sdbf | Target_30.sdbf | 268,273,270, 272, & 272 | 271 | 161,160,159, 158, & 158 | 159 | 1.70 |
| Reference_1000.sdbf | Target_30.sdbf | 435,395,397, 407, & 406 | 408 | 181,181,181, 181, &182 | 181 | 2.25 |
| Reference_2000.sdbf | Target_30.sdbf | 905,913,929, 927, & 920 | 919 | 315,313,316, 320, & 318 | 316 | 2.91 |

Table 7. Comparison of varying sizes of reference sets and the16GB target set

The biggest time difference between the CPU and the GPU for this comparison was attained when I compared reference_2000.sdbf and target_30.sdbf. It took the CPU implementation 15 minutes and 19 seconds to compare the set, while the GPU implementation took only 5 minutes and 16 seconds. This was a time difference of 10 minutes and 3 seconds.

Based on the result of the speedup (Avg. CPU Time divided by Avg. GPU Time) in Table 7, it is noted that GPU performs 1.29x, 1.65x, 1.70x, 2.25x, and 2.91x times faster than CPU when comparing reference_250.sdbf, reference_500.sdbf, reference_750.sdbf, reference_1000.sdbf, and reference_2000.sdbf to target_30.sdbf respectively. What this means is that as the reference sets digest size increase, I gained better GPU performance. Figure 5 illustrates the time differences between the CPU and the GPU for this comparison.

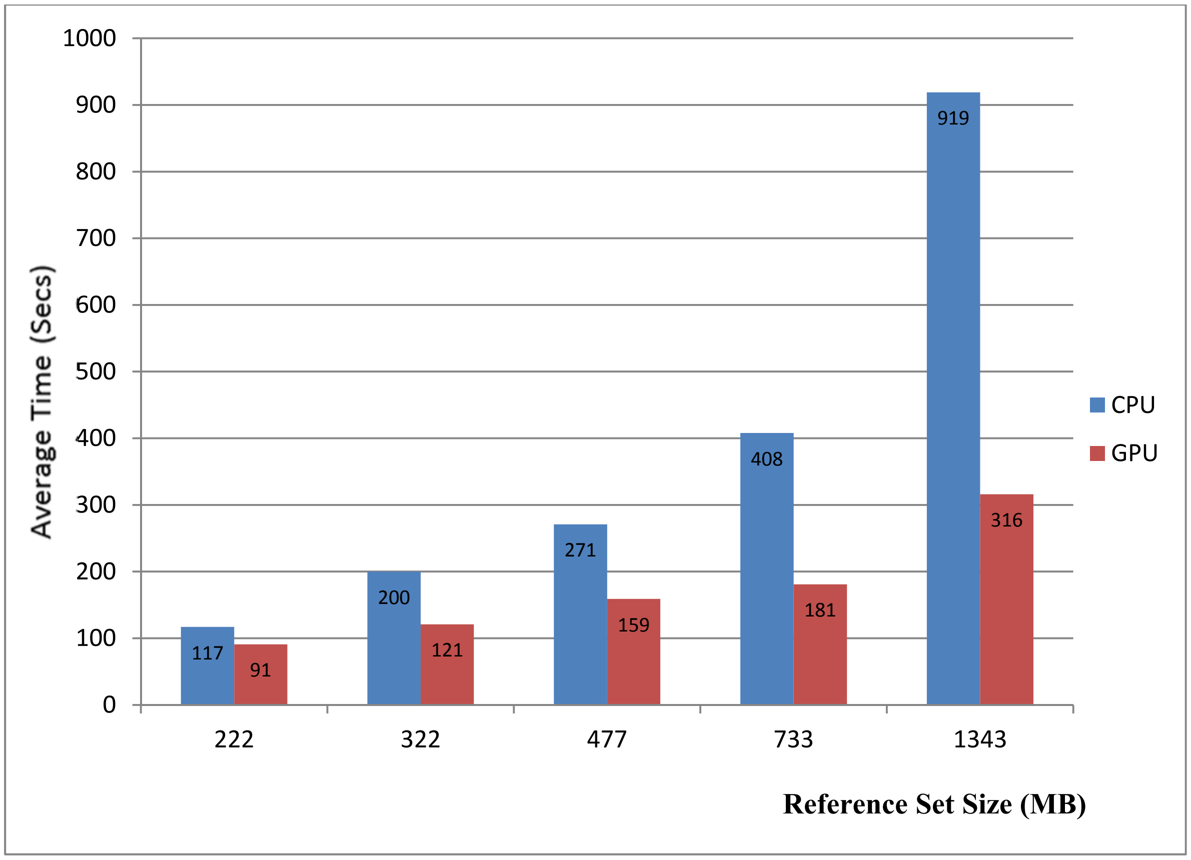


Figure 5. Differences in time required to compare varying sizes of reference sets and the16GB target set

The second comparison was carried out between each of the five reference sets and target_36.sdbf. As a reminder, target_36.sdbf contained 36,000 digests (607MB) and resulted from inputting 36,000 files (19GB) into *sdhash*. The biggest time difference between the CPU and GPU for this comparison was attained when I compared reference_2000.sdbf and target_36.sdbf. It took the CPU implementation 18 minutes and 49 seconds to compare the set, while GPU implementation took only 6 minutes and 24 seconds. This was a time difference of 12 minutes and 25 seconds. The results of these comparisons are shown in Table 8.

| Reference Sets Name | Target Set Name | CPU Time in Seconds (5 Runs) | Avg. CPU Time 5-runs (Secs) | GPU Time in Seconds (5 Runs) | Avg. GPU Time 5-runs (Secs) | Speedup |
|---|---|---|---|---|---|---|
| Reference_250.sdbf | Target_36.sdbf | 141,144,143,144, & 143 | 143 | 107,106,106, 108, & 107 | 107 | 1.34 |
| Reference_500.sdbf | Target_36.sdbf | 243,245,244,246, & 244 | 244 | 145,143,140, 148, & 146 | 144 | 1.69 |
| Reference_750_sdbf | Target_36.sdbf | 327,329,334,330, & 332 | 330 | 192,192,192, 192, & 192 | 192 | 1.72 |
| Reference_1000.sdbf | Target_36.sdbf | 530,532,529,524, & 529 | 529 | 215,217,217, 218, & 215 | 216 | 2.45 |
| Reference_2000.sdbf | Target_36.sdbf | 1135,1125,1125, 1123, & 1136 | 1129 | 263,260,257, 266, & 260 | 384 | 2.94 |

Table 8. Comparison of varying sizes of reference sets and the 19GB target set

Figure 6 illustrates the time differences between comparison of reference sets and 19GB target set on both CPU and GPU

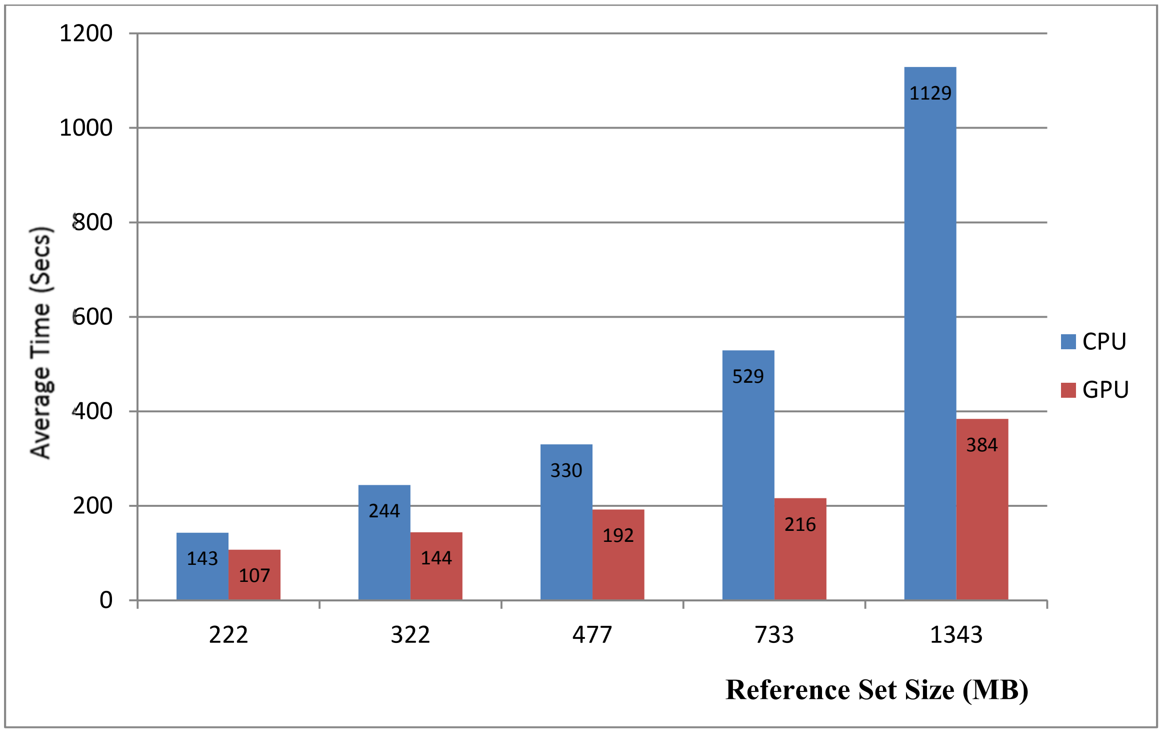
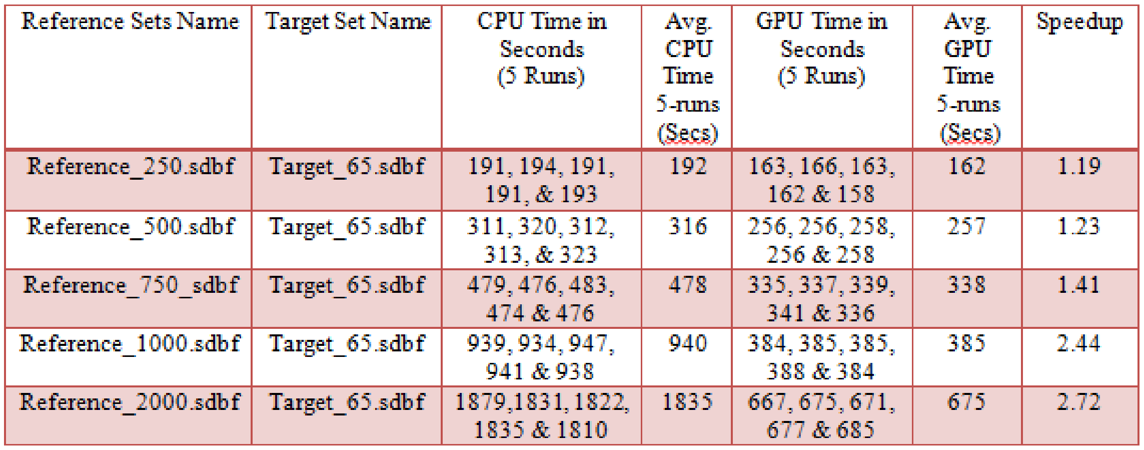


Figure 6. Differences in time required to compare varying sizes of reference sets to the 19GB target set

The third comparison was done between each of the five reference sets and target_65.sdbf. As a reminder, target_65.sdbf contained 65,000 digests (1.1GB), which resulted from inputting 65,000 files (34GB) into *sdhash*. The biggest time difference between the CPU and GPU for this comparison was achieved when I compared reference_2000.sdbf and target_65.sdbf. It took CPU implementation 30 minutes and 35 seconds to compare the set, while GPU implementation took 11 minutes and 15 seconds. This was a time difference of 19 minutes and 24 seconds. Table 9 shows the results.



| Reference Sets Name | Target Set Name | CPU Time in Seconds (5 Runs) | Avg. CPU Time 5-runs (Secs) | GPU Time in Seconds (5 Runs) | Avg. GPU Time 5-runs (Secs) | Speedup |
|---|---|---|---|---|---|---|
| Reference_250.sdbf | Target_65.sdbf | 191, 194, 191, 191, & 193 | 192 | 163, 166, 163, 162 & 158 | 162 | 1.19 |
| Reference_500.sdbf | Target_65.sdbf | 311, 320, 312, 313, & 323 | 316 | 256, 256, 258, 256 & 258 | 257 | 1.23 |
| Reference_750_sdbf | Target_65.sdbf | 479, 476, 483, 474 & 476 | 478 | 335, 337, 339, 341 & 336 | 338 | 1.41 |
| Reference_1000.sdbf | Target_65.sdbf | 939, 934, 947, 941 & 938 | 940 | 384, 385, 385, 388 & 384 | 385 | 2.44 |
| Reference_2000.sdbf | Target_65.sdbf | 1879,1831, 1822, 1835 & 1810 | 1835 | 667, 675, 671, 677 & 685 | 675 | 2.72 |

Table 9. Comparison of varying sizes of reference sets and the 34GB target set

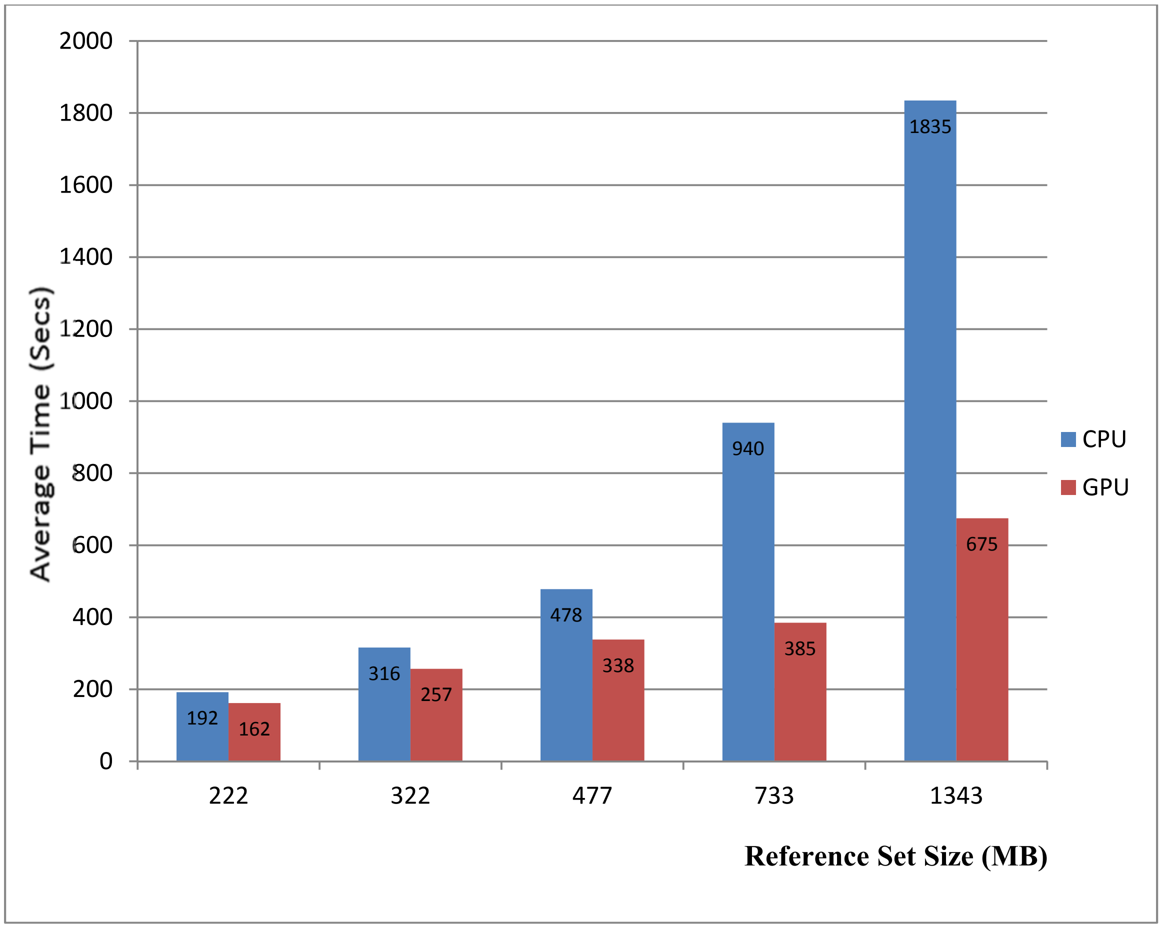


Figure 7. Differences in time required to compare varying sizes of reference sets and the 34GB target set

The fourth comparison was performed between each of the five reference sets and target_183.sdbf. As a reminder, target_183.sdbf contained 183,890 digests (2.4GB) and resulted from inputting 183,890 files (100GB) into *sdhash*. The biggest time difference between the CPU and GPU for this comparison was attained when I compared reference_2000.sdbf and target_183.sdbf. It took CPU implementation 73 minutes and 45 seconds to compare the set, while GPU implementation took only 29 minutes 21 seconds. This was a time difference of 45 minutes and 24 seconds. Table 10 displays the results.

| Reference Sets Name | Target Set Name | CPU Time in Seconds (5 Runs) | Avg. CPU Time 5-runs (Secs) | GPU Time in Seconds (5 Runs) | Avg. GPU Time 5-runs (Secs) | Speedup |
|---|---|---|---|---|---|---|
| Reference_250.sdbf | Target_183.sdbf | 678, 657, 638, 634 & 643 | 650 | 505, 508, 499, 510 & 498 | 504 | 1.29 |
| Reference_500.sdbf | Target_183.sdbf | 1017, 1014, 1011, 1010 & 1008 | 1012 | 689, 683, 677, 678 & 688 | 683 | 1.48 |
| Reference_750_sdbf | Target_183.sdbf | 1363, 1359, 1354, 1356 & 1362 | 1359 | 863, 858, 866, 861 & 868 | 863 | 1.57 |
| Reference_1000.sdbf | Target_183.sdbf | 2260, 2261, 2264, 2259 & 2265 | 2262 | 984, 985, 983, 984 & 984 | 984 | 2.30 |
| Reference_2000.sdbf | Target_183.sdbf | 4401, 4432, 4430, 4443 & 4420 | 4425 | 1744, 1782, 1766, 1757, & 1756 | 1761 | 2.51 |

Table 10. Comparison of varying sizes of reference sets and the 100GB target set

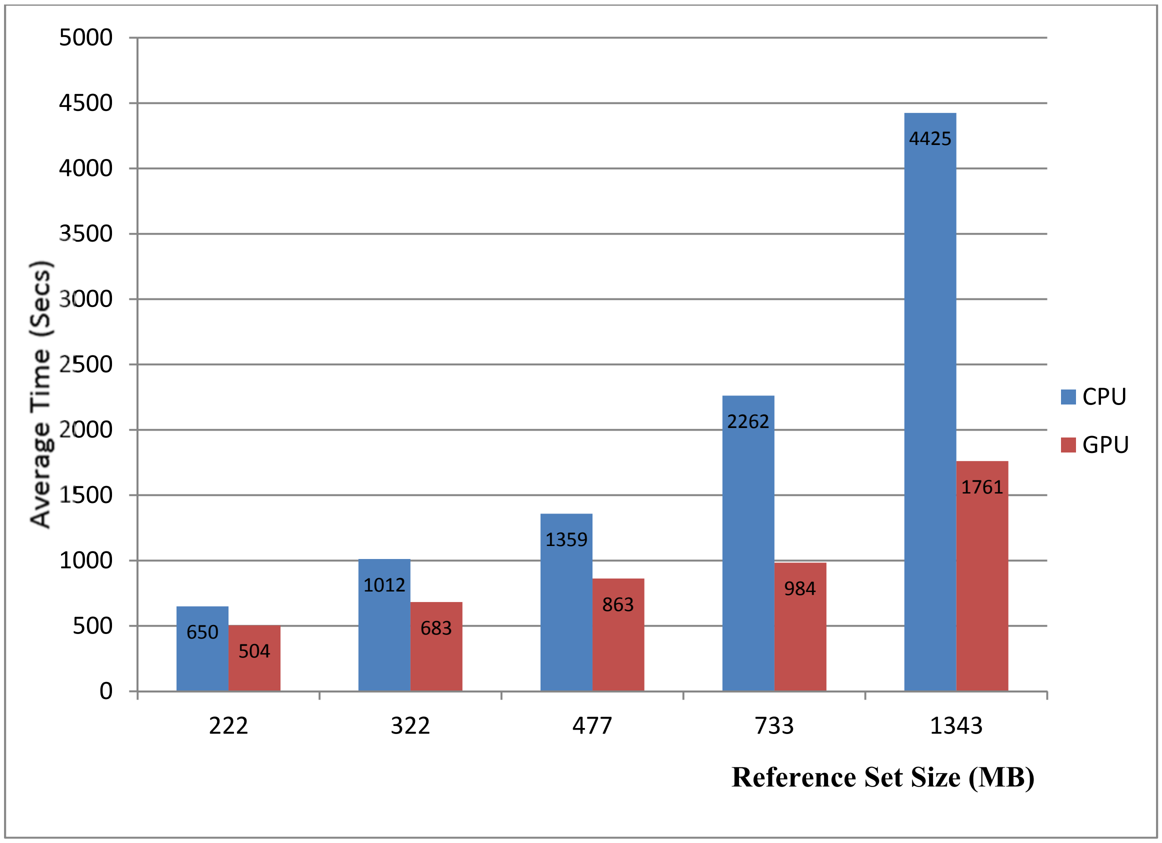
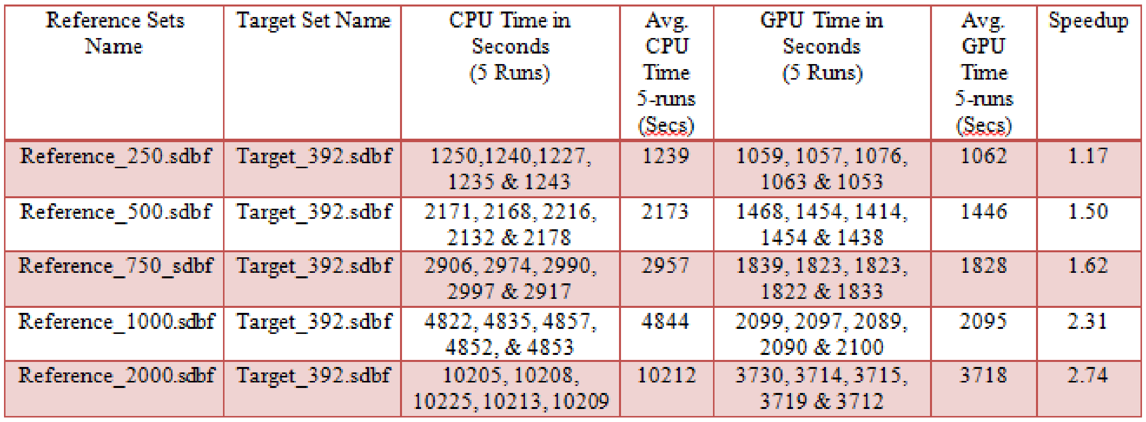


Figure 8. Differences in time required to compare varying sizes of reference sets and the 100GB target set

The last comparison was performed between each of the five reference sets and target_392.sdbf. As a reminder, target_392.sdbf contained 392,078 digests (5.1GB) and resulted from inputting 392,078 files (200GB) into *sdhash* algorithm. The biggest time difference between CPU and GPU was attained for this comparison when I compared reference_2000.sdbf and target_392.sdbf. It took CPU implementation 170 minutes and 12 seconds to compare the set, while GPU implementation took only 61 minutes and 58 seconds. This was a time difference of 108 minutes and 14seconds (1 hour 48 minutes and 14 seconds). Table 11 displays the results.



| Reference Sets Name | Target Set Name | CPU Time in Seconds (5 Runs) | Avg. CPU Time 5-runs (Secs) | GPU Time in Seconds (5 Runs) | Avg. GPU Time 5-runs (Secs) | Speedup |
|---|---|---|---|---|---|---|
| Reference_250.sdbf | Target_392.sdbf | 1250,1240,1227, 1235 & 1243 | 1239 | 1059, 1057, 1076, 1063 & 1053 | 1062 | 1.17 |
| Reference_500.sdbf | Target_392.sdbf | 2171, 2168, 2216, 2132 & 2178 | 2173 | 1468, 1454, 1414, 1454 & 1438 | 1446 | 1.50 |
| Reference_750_sdbf | Target_392.sdbf | 2906, 2974, 2990, 2997 & 2917 | 2957 | 1839, 1823, 1823, 1822 & 1833 | 1828 | 1.62 |
| Reference_1000.sdbf | Target_392.sdbf | 4822, 4835, 4857, 4852, & 4853 | 4844 | 2099, 2097, 2089, 2090 & 2100 | 2095 | 2.31 |
| Reference_2000.sdbf | Target_392.sdbf | 10205, 10208, 10225, 10213, 10209 | 10212 | 3730, 3714, 3715, 3719 & 3712 | 3718 | 2.74 |

Table 11. Comparison of varying sizes of reference sets and the 200GB target set

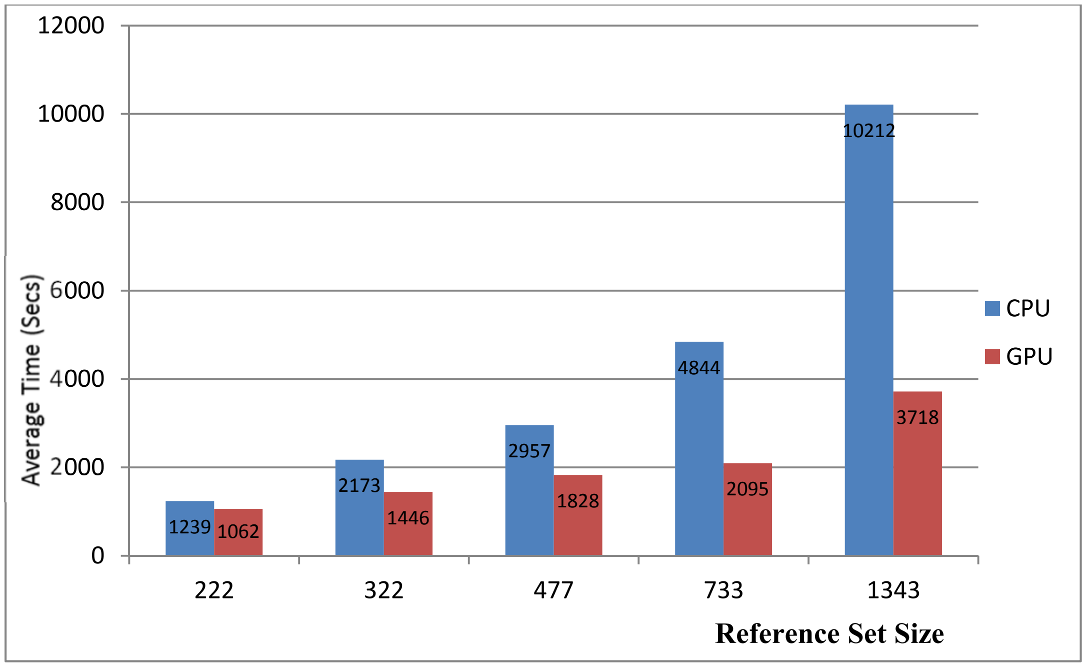


Figure 9. Differences in time required to compare varying sizes of reference sets and the 200GB target set

After all the comparisons, it became clear that the bigger the data sets to compare, the better the GPU performance. This was evidenced when I compared my biggest reference set, which was 2000 digests (1.3GB) and my biggest target set, which was 392,078 digests (200GB). In this particular comparison, it took CPU implementation 2 hours and 50 minutes to compare reference_2000.sdbf and target_392.sdbf; meanwhile, doing the same comparison in the GPU only took 1 hour and 2 minutes.

With small data sets, there were no significant time differences between CPU and GPU comparison. As seen on all the comparison tables and charts.

Even though the comparison time got better as I increased my data sets, I did not see a significant increase in the speedup. I wanted to keep increasing the reference set size to see if there will be a significant change in speedup, but the way the algorithm was written, I could not. I can only increase the target set size. As seen in Table 12 and Figure 10, the highest speedup was achieved when comparing the 1.3GB (2000 digest files) reference set and 19.3GB (36,000 digest files).

| | Target Sets | | | | |
|---|---|---|---|---|---|
| | 16GB | 19.3GB | 34GB | 100GB | 200GB |
| Ref = 222MB | 1.29 | 1.34 | 1.18 | 1.29 | 1.17 |
| Ref = 322MB | 1.65 | 1.69 | 1.23 | 1.48 | 1.50 |
| Ref= 477MB | 1.70 | 1.72 | 1.41 | 1.57 | 1.62 |
| Ref = 733MB | 2.25 | 2.45 | 2.44 | 2.30 | 2.31 |
| Ref = 1343MB | 2.91 | 2.94 | 2.72 | 2.51 | 2.74 |

Table 12. Speedup between CPU and GPU

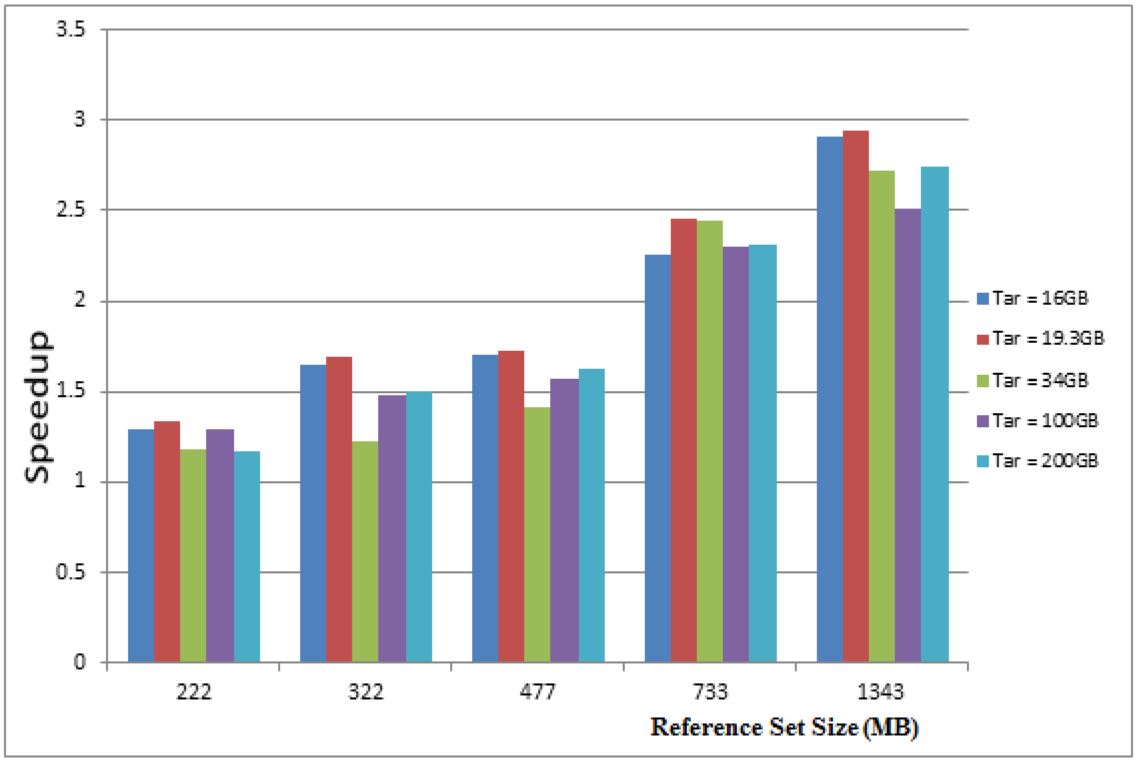


Figure 10. Chart showing speedup between CPU and GPU

# V. CONCLUSION

*Sdhash* is an approximate matching algorithm. This capstone examined the potential to use *sdhash* to look for active transfer of sensitive files over the network. Because of the computationally intensive nature of comparing digests, it has not been implemented as a way to protect sensitive data on networks. To this end, I demonstrated speed performance of *sdhash* on both the CPU and GPU. The results of this experiment showed that better performance is achieved with the GPU when comparing large data sets, which means GPU will perform well when large amounts of data are involved. The experiment also showed that there were no significant differences in time when comparing small digests in the CPU and GPU, which means if the amount of data to be compared is minimal, there is no need to incur extra expenses to implement the GPU.

The main contribution of this experiment is establishing feasibility of *sdhash* approximate matching in detecting data exfiltration over the network and determining which *sdhash* implementation is suitable for large networks such as DOD.

Based on the result of this experiment, I concluded that the CPU implementation of *sdhash* will be more suitable in a small- to medium-network environment or more suitable in an environment where sensitive data are less common; meanwhile the GPU implementation of *sdhash* will be more suitable for high network environment such as the DOD, where thousands of files may traverse the network in any given day. The GPU implementation of *sdhash* will also be suitable in an organization that deals with and processes a high volume of sensitive data.

The use of *sdhash* to compare sensitive files and files captured over the network is not going to be in real-time. It is meant more for batch processing, offline, of large amounts of data, as demonstrated in this experiment. The idea is to capture files traversing the network, maybe for a day, and then run *sdhash* algorithm against captured files offline to generate similarity digests that will be compared to similarity digests of sensitive files in order to determine if there were traces of sensitive file that left the network. In other words, it is more comparable to an intrusion detection mechanism rather than to an intrusion prevention mechanism.

For this experiment, I used the GovDocs corpus to simulate both sensitive files and files captured over the network; therefore for my future work, I will demonstrate this experiment on real network captures and real modified sensitive files that will be transferred and interleaved with regular network traffic. This will allow me the opportunity to measure both false positive and false negative rates of *sdhash*. I will also delve into how *sdhash* can be incorporated into GPU-based NIDS.

Even though *sdhash* can be a very useful tool in detecting exfiltration of sensitive file, it not going to completely stop exfiltration and leakage. There are still some ways any determined malicious insider can trick it if they were aware of its presence on the networks. Malicious insiders can encrypt the sensitive file before exfiltration, they can zip the file or embed it in another file (steganography).

Even though *sdhash* has some limitations, it still presents a better alternative to exact file matching, which can be easily tricked with character substitution and deletion. It is also a better alternative to cryptographic hashes, which will not detect exfiltration if a malicious insider changed a byte in sensitive file before exfiltration, owing to its "avalanche effect" property.

# APPENDIX A. GLOSSARY

Selected terms used in this paper are defined below.

**Algorithm:** Any well-defined computational procedure that takes some values as input and produce some value or set of values as output.

**Data-at-rest:** is defined as data stored in a persistence storage such as disk and tape

**Data-in-motion:** is defined as data being transferred between two nodes. Also called data-in-transit

**Device Control Module (DCM):** It provides the ability to restrict system access to peripheral devices such as thumb drives and other removable storage.

**ePolicy Orchestrator (ePO):** A management server responsible for collecting events, controlling policies, and maintaining updated content for end-point product modules on all HBSS clients.

**Event:** Any observable occurrence in a network or system.

**False Negative:** An alert that fails to indicate malicious activity is occurring.

**False Positive:** An alert that incorrectly indicates that malicious activity is occurring.

**Incident:** A violation of computer security policies, acceptable use policies, or standard security policy.

**Host Intrusion Prevention System (HIPS):** Software that automates the process of monitoring the events occurring in a computer system.

**McAfee Agent (MA):** The software agent on a host system that provides local management of all HBSS products installed on the host. The MA is utilized by ePolicy Orchestrator to coordinate communication of events, enforcement of policies, product deployment, content updating and management of each of the HBSS modules.

THIS PAGE INTENTIONALLY LEFT BLANK

# APPENDIX B. CPU INFORMATION

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                64
On-line CPU(s) list:   0-63
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             4
NUMA node(s):          8
Vendor ID:             AuthenticAMD
CPU family:            21
Model:                 1
Stepping:              2
CPU MHz:               1400.000
BogoMIPS:              4399.40
Virtualization:        AMD-V
L1d cache:             16K
L1i cache:             64K
L2 cache:              2048K
L3 cache:              6144K
NUMA node0 CPU(s):     0-7
NUMA node1 CPU(s):     8-15
NUMA node2 CPU(s):     16-23
NUMA node3 CPU(s):     24-31
NUMA node4 CPU(s):     32-39
NUMA node5 CPU(s):     40-47
NUMA node6 CPU(s):     48-55
NUMA node7 CPU(s):     56-63

====================
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 21
model           : 1
model name      : AMD Opteron(TM) Processor 6274
stepping        : 2
cpu MHz                 : 1400.000
cache size      : 2048 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
```

```
apicid          : 32
initial apicid  : 0
fpu             : yes
fpu_exception : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm
constant_tsc rep_good nonstop_tsc extd_apicid amd_dcm aperfmperf pni pclmulqdq
monitor ssse3 cx16 sse4_1 sse4_2 popcnt aes xsave avx lahf_lm cmp_legacy svm extapic
cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs xop skinit wdt lwp fma4
nodeid_msr topoext perfctr_core cpb npt lbrv svm_lock nrip_save tsc_scale vmcb_clean
flushbyasid decodeassists pausefilter pfthreshold
bogomips        : 4400.03
TLB size        : 1536 4K pages
clflush size    : 64
cache_alignment         : 64
address sizes   : 48 bits physical, 48 bits virtual
power management: ts ttp tm 100mhzsteps hwpstate cpb
```

# APPENDIX C. GPU INFORMATION

```
compute-8-17
    state = free
    np = 16
    properties = intel
    ntype = cluster
    status =
rectime=1424557697,varattr=,jobs=,state=free,netload=272785332869,gres=,loadave=1.
00,ncpus=16,physmem=132129940kb,availmem=129514976kb,totmem=132129940kb,i
dletime=266653,nusers=0,nsessions=0,uname=Linux compute-8-17 2.6.32-
431.20.3.el6.x86_64 #1 SMP Thu Jun 19 21:14:45 UTC 2014 x86_64,opsys=linux
    mom_service_port = 15002
    mom_manager_port = 15003
    gpus = 8
    gpu_status =
gpu[7] = gpu_id=0000:8A:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:8A:
00.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_util
ization=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=30 C.

gpu[6] = gpu_id=0000:89:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:89:0
0.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utili
zation=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A; gpu_temperature=28 C.

gpu[5] = gpu_id=0000:86:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:86:0
0.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utili
zation=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/Agpu_temperature=31 C.

gpu[4] = gpu_id=0000:85:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:85:0
0.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utili
zation=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=31 C.

gpu[3] = gpu_id=0000:09:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:09:0
0.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utili
zation=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=30 C.

gpu[2] = gpu_id=0000:08:00.0;gpu_product_name=GeForce GTX TITAN
Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:08:0
```

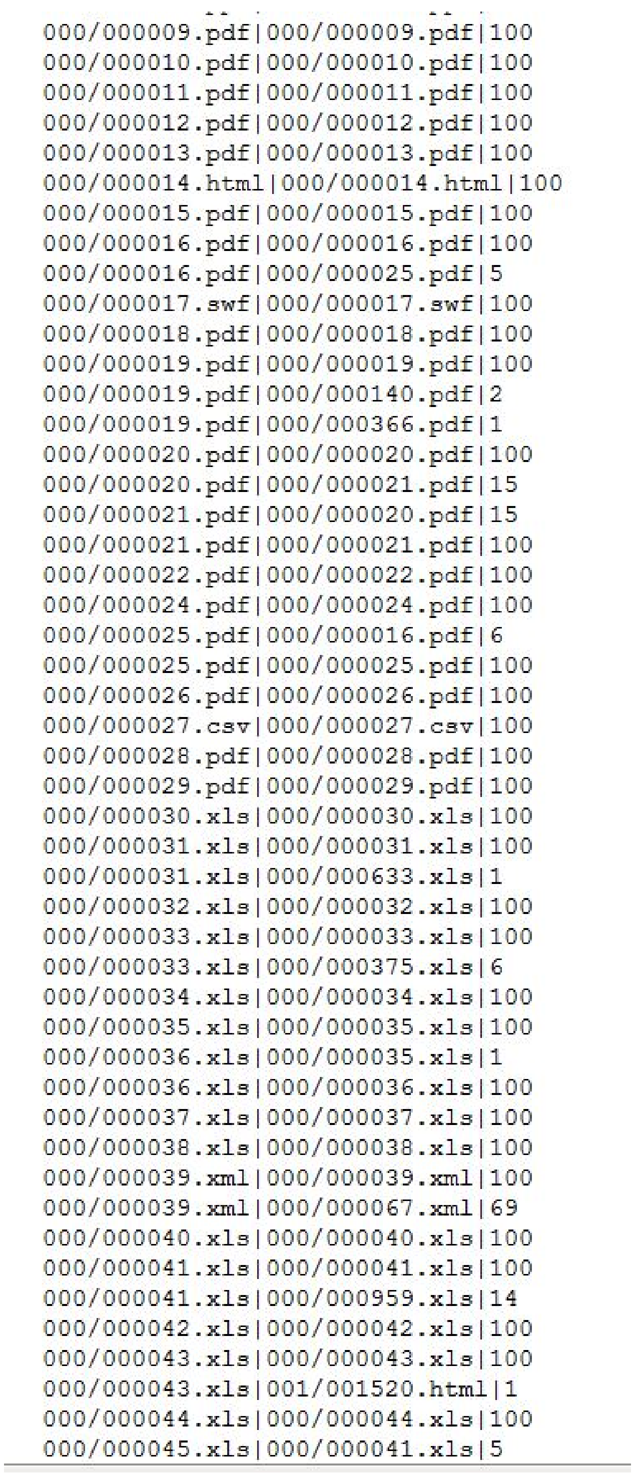
0.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utilization=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=29 C.

***gpu[1]*** = gpu_id=0000:05:00.0;gpu_product_name=GeForce GTX TITAN Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:05:00.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utilization=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=30 C.

***gpu[0]*** = gpu_id=0000:04:00.0;gpu_product_name=GeForce GTX TITAN Black;gpu_display=N/A;gpu_pci_device_id=100C10DE;gpu_pci_location_id=0000:04:00.0;gpu_fan_speed=26%;gpu_mode=Exclusive_Thread;gpu_state=Unallocated;gpu_utilization=N/A;gpu_memory_utilization=N/A;gpu_ecc_mode=N/A;gpu_temperature=34 C,driver_ver=340.29,timestamp=Sat Feb 21 14:28:16 2015

```
+------------------------------------------------------+
| NVIDIA-SMI 340.29     Driver Version: 340.29         |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX TIT...  Off  | 0000:04:00.0     N/A |                  N/A |
| 26%   34C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   1  GeForce GTX TIT...  Off  | 0000:05:00.0     N/A |                  N/A |
| 26%   30C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   2  GeForce GTX TIT...  Off  | 0000:08:00.0     N/A |                  N/A |
| 26%   28C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   3  GeForce GTX TIT...  Off  | 0000:09:00.0     N/A |                  N/A |
| 26%   30C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   4  GeForce GTX TIT...  Off  | 0000:85:00.0     N/A |                  N/A |
| 26%   30C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   5  GeForce GTX TIT...  Off  | 0000:86:00.0     N/A |                  N/A |
| 26%   30C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   6  GeForce GTX TIT...  Off  | 0000:89:00.0     N/A |                  N/A |
| 26%   28C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A    E. Thread |
+-------------------------------+----------------------+----------------------+
|   7  GeForce GTX TIT...  Off  | 0000:8A:00.0     N/A |                  N/A |
| 26%   30C    P0    N/A /  N/A |     15MiB /  6143MiB |     N/A      Default |
+-------------------------------+----------------------
```

# APPENDIX D. SAMPLE COMPARISON OUTPUT



```
000/000009.pdf|000/000009.pdf|100
000/000010.pdf|000/000010.pdf|100
000/000011.pdf|000/000011.pdf|100
000/000012.pdf|000/000012.pdf|100
000/000013.pdf|000/000013.pdf|100
000/000014.html|000/000014.html|100
000/000015.pdf|000/000015.pdf|100
000/000016.pdf|000/000016.pdf|100
000/000016.pdf|000/000025.pdf|5
000/000017.swf|000/000017.swf|100
000/000018.pdf|000/000018.pdf|100
000/000019.pdf|000/000019.pdf|100
000/000019.pdf|000/000140.pdf|2
000/000019.pdf|000/000366.pdf|1
000/000020.pdf|000/000020.pdf|100
000/000020.pdf|000/000021.pdf|15
000/000021.pdf|000/000020.pdf|15
000/000021.pdf|000/000021.pdf|100
000/000022.pdf|000/000022.pdf|100
000/000024.pdf|000/000024.pdf|100
000/000025.pdf|000/000016.pdf|6
000/000025.pdf|000/000025.pdf|100
000/000026.pdf|000/000026.pdf|100
000/000027.csv|000/000027.csv|100
000/000028.pdf|000/000028.pdf|100
000/000029.pdf|000/000029.pdf|100
000/000030.xls|000/000030.xls|100
000/000031.xls|000/000031.xls|100
000/000031.xls|000/000633.xls|1
000/000032.xls|000/000032.xls|100
000/000033.xls|000/000033.xls|100
000/000033.xls|000/000375.xls|6
000/000034.xls|000/000034.xls|100
000/000035.xls|000/000035.xls|100
000/000036.xls|000/000035.xls|1
000/000036.xls|000/000036.xls|100
000/000037.xls|000/000037.xls|100
000/000038.xls|000/000038.xls|100
000/000039.xml|000/000039.xml|100
000/000039.xml|000/000067.xml|69
000/000040.xls|000/000040.xls|100
000/000041.xls|000/000041.xls|100
000/000041.xls|000/000959.xls|14
000/000042.xls|000/000042.xls|100
000/000043.xls|000/000043.xls|100
000/000043.xls|001/001520.html|1
000/000044.xls|000/000044.xls|100
000/000045.xls|000/000041.xls|5
```

THIS PAGE INTENTIONALLY LEFT BLANK

# LIST OF REFERENCES

[1] S. Kerner. (2015, Feb. 11). HP security study finds internal defenses lacking. [Online]. Available: http://www.eweek.com/security/hp-security-study-finds-internal-defenses-lacking.html

[2] D. Cappelli et al. *The CERT Guide to Insider Threat*, 2nd ed. Westford, MA: Pearson Education, 2014, preface xx.

[3] An executive's guide to 2013 data breach trends. (2015, Jan. 9). Open Security Foundation. [Online]. Available: https://www.riskbasedsecurity.com/reports/2013-DataBreachQuickView.pdf

[4] 2013 U.S. state of cybercrime survey. (2015, Jan. 9) CERT Program, Carnegie Mellon University - Software Engineering Institute. [Online]. Available: http://resources.sei.cmu.edu/asset_files/Presentation/2013_017_101_58739.pdf

[5] J. Roman. (2015, Jan. 10). Insider risks: What have we learned? [Online]. Available: http://www.bankinfosecurity.com/insider-risks-what-have-we-learned-a-7195

[6] J. Roman. (2015, Jan. 10). Morgan Stanley: Insider stole data. [Online]. Available: http://www.bankinfosecurity.com/morgan-stanley-insider-stole-data-a-7750

[7] A. Sternstein. (2015, Feb. 09). Pentagon spent millions to counter insider threats after wikileaks fiasco. [Online]. Available: http://www.nextgov.com/cybersecurity/2013/07/defense-spent-millionscounterinsider-threats-after-wikileaks-fiasco/65843

[8] L. Park. (2014, Dec. 18). Data breach trends. [Online]. Available: http://www.symantec.com/connect/blogs/data-breach-trends

[9] M. Cruz. (2015, Jan. 10). Data exfiltration in targeted attacks. [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/data-exfiltration-in-targeted-attacks/

[10] Hbss. (n.d.). [Online]. Available: http://www.disa.mil/services/cybersecurity/hbss. Accessed Jan. 10, 2015

[11] M. Hamilton. (2014, Jan. 17). Defining data mapping and data loss prevention technology for financial firms. [Online]. Available: http://www.eci.com/blog/13-defining-data-mapping-and-data-loss-prevention

[12] R. Trezza. (2011, Jul. 26). McAfee data loss prevention (DLP). [YouTube video]. Available: https://www.youtube.com/watch?v=TXYNNSaMxsI. Accessed Jan. 15, 2015.

[13] A. Apostolico et al., *Pattern Matching Algorithms*. New York, NY: Oxford University Press, 1997, pp. 1–2.

[14] N Desai. Increasing performance in high speed NIDS: A look at Snort's internals. [Online]. Available: http://www.linuxsecurity.com/resource_files/intrusion_detection/Increasing_Performance_in_High_Speed_NIDS.pdf

[15] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev*., 31(2): 249–260, March 1987. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5390135

[16] Stoimen. (2012, Apr. 2). Computer algorithms: Rabin-Karp string searching. [Online]. Available: http://www.stoimen.com/blog/2012/04/02/computer-algorithms-rabin-karp-string-searching

[17] D. E. Knuth et al. (1977, June). Fast pattern matching in strings. *Siam Journal on Computing*, 6(2): 322–350. Available: http://epubs.siam.org/doi/pdf/10.1137/0206024

[18] R. S. Boyer and J. S. Moore. (1977, October). A fast string searching algorithm. *CACM*, 20(10): 762–772. [Online]. Available: http://www.cs.utexas.edu/users/moore/publications/fstrpos.pdf

[19] S. Vanveerdeghem. (2015, Feb. 21). Introduction to regular expressions for IPS. [Online]. Available: https://supportforums.cisco.com/blog/149481/introduction-regular-expressions-ips

[20] National Institute of Standards and Technology. (2012, March). *Secure Hash Standard (SHS)*, FIPS Pub 180-4, Gaithersburg, MD. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf

[21] Metadata for authenticity: Hash functions and digital signatures. (n.d.). [Online]. Available: http://www.paradigm.ac.uk/workbook/metadata/authenticity-fixity.html. Accessed Feb. 24, 2015.

[22] F. Breitinger, G. Barbara, M. Michael, and R. Vassil. (2014, May). *Approximate Matching: Definition and Terminology*, NIST Special Publication 800-168. [Online]. Available: http://csrc.nist.gov/publications/drafts/800-168/sp800_168_draft.pdf

[23] V. Gupta. (2013, Aug. 20). File detection in network traffic using approximate matching. [Online]. Available: http://www.divaportal.org/smash/get/diva2:651455/FULLTEXT01.pdf

[24] J Kornblum. (2006). Identifying almost identical files using context triggered piecewise hashing. [Online]. Available: http://dfrws.org/2006/proceedings/12Kornblum.pdf

[25] V. Roussev. (2010). Data fingerprinting with similarity digests. [Online]. Available: http://roussev.net/pubs/2010-IFIP--sdhash-design.pdf

[26] V. Roussev and C. Quates. (2012, Sep.). Content triage with similarity digests: The M57 case study. [Online]. Available: http://www.dfrws.org/2012/proceedings/DFRWS2012-7.pdf

[27] F. Breitinger and H. Baier. (2015, Feb. 16). Similarity Preserving Hashing: Eligible Properties and a new Algorithm MRSH-v2. [Online]. Available: https://www.dasec.h-da.de/wpcontent/uploads/2012/11/2012_10_Breitinger-_Baier_ICDF2C.pdf.

[28] F. Breitinger and V. Roussev. (2015, Feb.16). *Automated Evaluation of Approximate Matching Algorithms on Real Data*. [Online]. Available: http://roussev.net/pubs/2014-DFRWS-EU--alcs.pdf

[29] V. Roussev and C. Quates. (2013, Aug. 6). [Online]. Available: http://roussev.net/sdhash/tutorial/sdhash-tutorial.pdf. Accessed Feb. 9, 2015.

[30] M. R. McCarrin, “Exploration and validation of the sdhash parameter space,” M.S. thesis, Dept. Comp. Sci., Naval Postgraduate Sch., Monterey, CA, 2013.

[31] GEFORCE GTX Titan Black. (n.d.). [Online]. Available: http://www.nvidia.com/gtx-700-graphics-cards/gtx-titan-black/. Accessed Feb. 21, 2015.

[32] What is CUDA? (n.d.). [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html. Accessed Mar. 3, 2015.

THIS PAGE INTENTIONALLY LEFT BLANK

# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California